直播系统开发

基于Nginx与Nginx-rtmp-module

卓晓科技技术团队◎编著

中国工信出版集团
电子工业出版社

直播系统开发
基于Nginx与Nginx-rtmp-module
卓晓科技技术团队◎编著

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

直播系统开发
基于Nginx与Nginx-rtmp-module
卓晓科技技术团队◎编著

直播系统开发
基于Nginx与Nginx-rtmp-module
卓晓科技技术团队◎编著

直播系统开发
基于Nginx与Nginx-rtmp-module
卓晓科技技术团队◎编著

目　录

第 1 章

Nginx 基础

近几年，直播行业越来越火爆，本书主要介绍开源的直播软件——Nginx-rtmp-module。Nginx-rtmp-module 依赖于 Nginx，以第三方模块的方式提供直播功能。本章主要介绍 Nginx，包括什么是 Nginx，为什么选择 Nginx，在特定的环境下如何安装和配置 Nginx，以及如何使用 Nginx，最后介绍 Nginx 模块的概念。

1.1 Nginx 概述及作用

Nginx 同 Apache、Tomcat 一样，是一种服务器软件。它是一个高性能的 HTTP 和反向代理服务器，同时也是一个 IMAP/POP3/SMTP 代理服务器。因此，使用 Nginx 可以搭建网站，也可以实现负载均衡的功能，还可以作为邮件代理服务器来接收和发送邮件。Nginx 1.9.0 以后的版本还可以作为通用的 TCP/UDP 代理服务器，也可以提供一定的缓存服务功能。

1.1.1 可作为 Web 服务器

Nginx 还是一个高性能的 HTTP Web 服务器（Web 服务器还有 Apache、IIS 等），它包含了基本的 HTTP 功能和拓展功能，可以先通过动态/静态内容分离，而后为静态内容（HTML/CSS/JaveScript/图片等）提供 HTTP 访问功能；而动态内容可以整合代理模块，代理给上游服务器，以支持对外部程序的直接调用或者解析，如 FastCGI 支持 PHP。

1.1.2 可作为反向代理服务器

代理服务器分为正向代理服务器和反向代理服务器。

1. 正向代理服务器

正向代理服务器是一个位于客户端与原始服务器之间的服务器。为了从原始服务器中取得数据，客户端向代理服务器发送请求并指定目标（原始服务器），然后，代理服务器向原始服务器转交请求，并将获得的内容返回客户端。

正向代理服务器一般作用在客户端，并且在客户端需要进行相关配置，如图 1-1 所示。

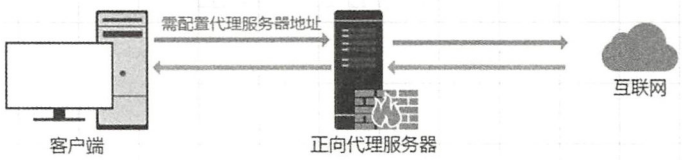


图 1-1

2. 反向代理服务器

反向代理服务器作用在服务器端，它在服务器端接收互联网中的连接请求，然后将请求转发给内部网络中的服务器，并将从服务器中得到的结果返回给互联网中请求连接的客户端，如图 1-2 所示。

反向代理对外是透明的，在客户端不需要任何配置，所以，访问者并不知道自己访问的是一个反向代理服务器。

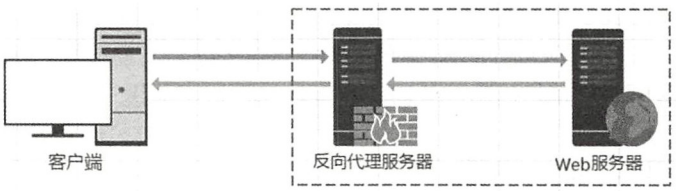


图 1-2

Nginx 就是一个反向代理服务器。

反向代理服务器针对 Web 服务器提供加速功能，所有外部网络要访问服务器的请求都要通过它。反向代理服务器负责接收客户端的请求，然后到源服务器上获取内容，把内容返回给用户，并把内容保存在本地中，以便日后再收到同样的信息请求时，将本地缓存中的内容直接发给用户，以减少后端 Web 服务器的压力，提高响应速度。因此，Nginx 还具有缓存功能。

3．反向代理服务器实现负载均衡

Nginx 可通过反向代理服务器来实现负载均衡，以优化网站的负载，如图 1-3 所示。

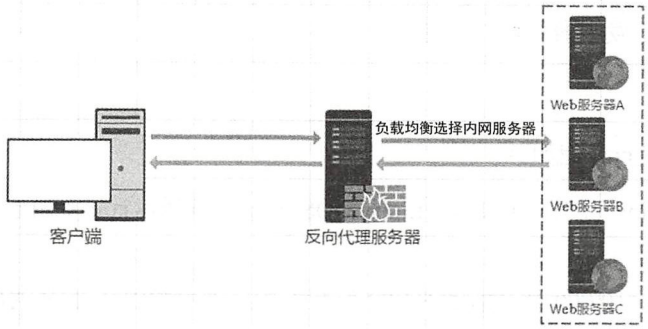


图 1-3

1.1.3　可作为邮件代理服务器

Nginx 可被部署成邮件代理服务器，最早开发 Nginx 的目的之一就是将其作为邮件代理服务器。

1.2　为什么选择 Nginx

Nginx 有着高并发、性能好和占用内存少等特点，其安装简单，配置文件简洁，启动容易，能长时间不间断运行，还能在不间断服务的情况下升级软件版本，而且成本低。这些优点使得 Nginx 的应用越来越普遍。

1．高并发、性能好、占用内存少和稳定

作为 Web 服务器，相比 Apache，Nginx 占用内存更少，支持的并发连接更多，使用效率更高，并且 Nginx 要比 Apache 更“轻量”，性能更好。

2．功能强大

Nginx 提供了大量的功能模块，支持诸多特性，应用场景也多，可作为 Web 服务器、反向代理服务器，也可作为邮件服务器等。

3．拓展性高

Nginx 的模块化设计极具拓展性，它完全是由多个不同功能、不同层次、不同类型且耦合

度极低的模块组成的。因此，当对某一个模块进行缺陷修复或升级时，可以专注于模块自身，而不会影响其他模块。

这种低耦合度的设计，使得 Nginx 具有数量庞大的第三方模块。当然，这些公开的第三方模块也如 Nginx 官方发布的模块一样易用。

4．其他优点

Nginx 的其他优点介绍如下。

- 跨平台：Nginx 可以在 UNIX、Linux、OS 系统中编译运行，而且也有 Windows 的移植版本。
- 占用内存小：10 000 个非活动 HTTP 保持连接，占用大约 2.5MB 的内存。
- 配置/操作简单：Nginx 安装简单，配置文件简洁，易上手。
- 网络依赖性低：理论上只要能够通过 ping 就可以实施负载均衡，而且可以有效区分内网、外网流量。
- 支持内置服务器检测：Nginx 能够根据服务器处理页面返回的状态码、超时信息等，检测服务器是否出现故障，并及时返回错误的请求，重新提交到其他节点上。

1.3 安装 Nginx

Nginx 可以在不同的操作系统、不同的环境中安装。本节以 CentOS 6.9 操作系统为例，介绍 Nginx 的安装和相关配置。

使用 Yum 安装 rpm 包的方式比编译安装的方式简单很多，其默认会安装许多模块，但缺点是以后再安装第三方模块时比较麻烦，所以，这里使用编译安装的方式安装 Nginx。

1.3.1 选择安装版本

在 Nginx 官网中可下载 Nginx 安装包，其中提供了 3 个版本：Mainline version、Stable version 和 Legacy versions。

Mainline version 是 Nginx 目前在主力研发的版本。Stable version 是最新的稳定版本，是生产环境中建议使用的版本。Legacy versions 是稳定的老版本。

这里选择 Stable version 版本：nginx-1.12.2.tar.gz。安装环境是 CentOS 6.9。因为在安装过程所执行的命令需要 root 权限，所以，这里选择使用 root 用户安装。

1.3.2　编译安装 Nginx

1. 准备工作

安装依赖包：gcc、g++。

安装必要的库：zlib、pcre、openssl。

源码编译依赖 gcc 环境，并且部分 Nginx 模块依赖于以上 3 个库，如果没有安装这 3 个库，则需要先安装。

2. 下载解压

将安装包下载到指定目录下并解压。

```
cd /usr/local/
wget http://nginx.org/download/nginx-1.12.2.tar.gz
tar -zxvf nginx-1.12.2.tar.gz
```

3. 配置

使用 configure 命令进行配置。它定义了系统的各个方面配置，包括 Nginx 允许用于连接处理的方法，并且最终创建了一个 Makefile 文件。

```
cd nginx-1.12.2
./configure --help
```

其中“./configure --help”命令能列出大部分常用模块和编译选项，其中部分内容如图 1-4 所示。

```
--help                             print this message

--prefix=PATH                      set installation prefix
--sbin-path=PATH                   set nginx binary pathname
--modules-path=PATH                set modules path
--conf-path=PATH                   set nginx.conf pathname
--error-log-path=PATH              set error log pathname
--pid-path=PATH                    set nginx.pid pathname
--lock-path=PATH                   set nginx.lock pathname

--user=USER                        set non-privileged user for
                                   worker processes
--group=GROUP                      set non-privileged group for
                                   worker processes

--build=NAME                       set build name
--builddir=DIR                     set build directory

--with-select_module               enable select module
--without-select_module            disable select module
--with-poll_module                 enable poll module
--without-poll_module              disable poll module
```

图 1-4

其中以--without 开头的选项都是默认安装的，以 PATH 结尾的选项需要手动指定依赖库源码目录。

（1）配置选项说明。

下面具体介绍一些常见的配置选项。

- --prefix=PATH：设置 Nginx 的安装目录，默认为/usr/local/nginx。
- --sbin-path=PATH：设置 Nginx 可执行文件的名称，默认为 prefix/sbin/nginx。
- --conf-path=PATH：设置 nginx.conf 配置文件的名称。Nginx 允许使用不同的配置文件启动服务，通过在命令行参数中指定要使用的配置文件，默认为 prefix/conf/nginx.conf。
- --pid-path=PATH：设置存储主进程 ID 的文件，默认为 prefix/logs/nginx.pid。安装后也可在 nginx.conf 中使用 pid 命令更改。
- --error-log-path=PATH：设置主要错误、警告和诊断文件。安装后，可以使用 error_log 命令在 nginx.conf 配置文件中更改文件名，默认为 prefix/logs/error.log。
- --http-log-path=PATH：设置 HTTP 服务器的主要请求日志文件。安装之后，可以使用 access_log 命令在 nginx.conf 配置文件中更改文件名，默认为 prefix/logs/access.log。
- --with-http_ssl_module：可以构建一个将 HTTPS 协议支持添加到 HTTP 服务器中的模块。该模块不是默认生成的。openssl 库需要构建和运行这个模块。
- --with-pcre=PATH：将路径设置为 pcre 库的来源。
- --with-zlib=PATH：将路径设置为 zlib 库的来源。

更为详细的配置选项说明请参考 Nginx 官网中的文档。

（2）配置命令。

```
./configure
```

这里都是选择默认配置，Nginx 将默认被安装到/usr/local/nginx 目录下。执行命令后部分结果如图 1-5 所示。

```
Configuration summary
  + using system PCRE library
  + OpenSSL library is not used
  + using system zlib library

  nginx path prefix: "/usr/local/nginx"
  nginx binary file: "/usr/local/nginx/sbin/nginx"
  nginx modules path: "/usr/local/nginx/modules"
  nginx configuration prefix: "/usr/local/nginx/conf"
  nginx configuration file: "/usr/local/nginx/conf/nginx.conf"
  nginx pid file: "/usr/local/nginx/logs/nginx.pid"
  nginx error log file: "/usr/local/nginx/logs/error.log"
  nginx http access log file: "/usr/local/nginx/logs/access.log"
  nginx http client request body temporary files: "client_body_temp"
  nginx http proxy temporary files: "proxy_temp"
  nginx http fastcgi temporary files: "fastcgi_temp"
  nginx http uwsgi temporary files: "uwsgi_temp"
  nginx http scgi temporary files: "scgi_temp"
```

图 1-5

4. 编译安装

```
make && make install
```

5. 验证是否安装成功

可以通过查看 Nginx 的版本信息来验证其是否安装成功。

```
/usr/local/nginx/sbin/nginx -v
```

如果安装成功，则会显示 Nginx 的版本信息，如图 1-6 所示。

```
[root@Nginx local]# /usr/local/nginx/sbin/nginx -v
nginx version: nginx/1.12.2
```

图 1-6

6. 修改配置文件

在安装 Nginx 的配置文件“nginx.conf”时，如果没有指定路径，则默认放在 /usr/local/nginx/conf 目录下，1.5 节会专门介绍 nginx.conf 文件中的相关配置。

```
vi /usr/local/nginx/conf/nginx.conf
```

7. 验证配置文件的正确性

如果修改了配置文件，则在启动 Nginx 之前，最好先检查一下配置文件是否正确，以免在重启 Nginx 之后出现错误，影响服务器的稳定运行，具体执行命令如下：

```
/usr/local/nginx/sbin/nginx -t
```

如果配置文件被正确执行，则结果如图 1-7 所示。

```
[root@Nginx /]# nginx -t
nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful
```

图 1-7

1.3.3　配置防火墙

安装好 Nginx 之后，需要配置防火墙，开启 80 端口。如果不开启 80 端口，则防火墙会阻止外网访问 80 端口，从而我们就无法访问 Nginx 的配置网站。

1. 防火墙相关操作

下面介绍几个与防火墙相关的命令：

```
#查看防火墙状态
service iptables status
```

```
#启动防火墙
service iptables start
#关闭防火墙
service iptables stop
#重启防火墙
service iptables restart
```

2. 配置防火墙

修改防火墙配置：vi /etc/sysconfig/iptables。

添加配置项：A INPUT -m state --state NEW -m tcp -p tcp --dport 80 -j ACCEPT。

重启防火墙：service iptables restart。

3. 启动 Nginx 服务

```
/usr/local/nginx/sbin/nginx
```

4. 查看 Nginx 进程信息

启动 Nginx 之后，便可以使用以下命令查看 Nginx 进程信息。

```
ps -ef | grep nginx
```

命令运行结果如图 1-8 所示。

```
[root@Nginx local]# ps -ef | grep nginx
root      3182     1  0 Jan24 ?        00:00:00 nginx: master process nginx
nobody    3187  3182  0 Jan24 ?        00:00:00 nginx: worker process
root      5833  5793  0 11:24 pts/0    00:00:00 grep nginx
```

图 1-8

其中，master process 对应的是主进程，3182 是主进程号，worker process 是工作进程。

Nginx 有一个主进程和多个工作进程。主进程主要用于读取和评估配置，并维护工作进程。工作进程是对请求进行实际处理。Nginx 使用基于事件的模型和依赖操作系统的机制来高效地在工作进程之间分配请求。工作进程的数量在配置文件中定义。

5. 测试

（1）测试 80 端口。

```
netstat -ntulp|grep 80
```

执行结果如图 1-9 所示。

```
[root@Nginx ~]# netstat -ntulp|grep 80
tcp        0      0 0.0.0.0:80                  0.0.0.0:*                   LISTEN      1945/nginx
```

图 1-9

（2）浏览器访问测试。

用浏览器访问地址：http://ip:80，其中“ip”是 Nginx 服务器的 IP 地址。访问结果如图 1-10 所示。

图 1-10

6．关闭 Nginx 服务

停止进程：kill -QUIT 主进程号。

快速停止：kill -TERM 主进程号。

强制停止：pkill -9 nginx。

1.3.4　加入自启动和系统服务

虽然可以用命令行对 Nginx 进行开启、关闭等各种操作，但毕竟不是很方便。可以配置 Nginx 到系统服务器中，从而可以通过 service 命令来启动和关闭服务。也可以将 Nginx 设为开机自启动，那么，在每次重启服务器之后就不用手动开启 Nginx 服务了，非常方便。

1．创建脚本文件

在/etc/init.d 目录下创建一个名为“nginx”的脚本文件，文件内容如下：

```
#!/bin/sh
#
# nginx - this script starts and stops the nginx daemon
#
# chkconfig:   - 85 15
# description: Nginx is an HTTP(S) server, HTTP(S) reverse \
#              proxy and IMAP/POP3 proxy server
# processname: nginx
```

```
# config:      /usr/local/nginx/conf/nginx.conf
# config:      /etc/sysconfig/nginx
# pidfile:     /var/run/nginx.pid

# Source function library.
. /etc/rc.d/init.d/functions

# Source networking configuration.
. /etc/sysconfig/network

# Check that networking is up.
[ "$NETWORKING" = "no" ] && exit 0

nginx="/usr/local/nginx/sbin/nginx"
prog=$(basename $nginx)

NGINX_CONF_FILE="/usr/local/nginx/conf/nginx.conf"

[ -f /etc/sysconfig/nginx ] && . /etc/sysconfig/nginx

lockfile=/var/lock/subsys/nginx

start() {
    [ -x $nginx ] || exit 5
    [ -f $NGINX_CONF_FILE ] || exit 6
    echo -n $"Starting $prog: "
    daemon $nginx -c $NGINX_CONF_FILE
    retval=$?
    echo
    [$retval -eq 0 ] && touch $lockfile
    return $retval
}

stop() {
    echo -n $"Stopping $prog: "
    killproc $prog -QUIT
    retval=$?
    echo
    [ $retval -eq 0 ] && rm -f $lockfile
    return $retval
killall -9 nginx
}

restart() {
    configtest || return $?
    stop
    sleep 1
```

```
    start
}

reload() {
    configtest || return $?
    echo -n $"Reloading $prog: "
    killproc $nginx -HUP
RETVAL=$?
    echo
}

force_reload() {
    restart
}

configtest() {
$nginx -t -c $NGINX_CONF_FILE
}

rh_status() {
    status $prog
}

rh_status_q() {
    rh_status >/dev/null 2>&1
}

case "$1" in
    start)
        rh_status_q && exit 0
    $1
        ;;
    stop)
        rh_status_q || exit 0
        $1
        ;;
    restart|configtest)
        $1
        ;;
    reload)
        rh_status_q || exit 7
        $1
        ;;
    force-reload)
        force_reload
        ;;
    status)
```

```
            rh_status
            ;;
        condrestart|try-restart)
            rh_status_q || exit 0
                ;;
        *)
          echo $"Usage: $0
{start|stop|status|restart|condrestart|try-restart|reload|force-reload|configtest}"
            exit 2
    esac
```

要根据实际安装路径，修改脚本中的以下两个配置选项：

将 nginx="/usr/local/nginx/sbin/nginx"修改成 Nginx 执行程序的路径。

将 NGINX_CONF_FILE="/usr/local/nginx/conf/nginx.conf"修改成配置文件的路径。

2．设置执行权限

要给脚本添加执行权限，不然执行的时候会报错：permission denied。

```
chmod 755 /etc/init.d/nginx
```

3．执行

```
#启动 Nginx
service nginx start
#关闭 Nginx
service nginx stop
#重启 Nginx
service nginx reload
```

4．加入开机自启动

```
#显示开机自动启动的服务
chkconfig --list
#将 Nginx 服务加入 chkconfig 管理列表
chkconfig --add nginx
#设置终端模式开机启动
chkconfig nginx on
#重启计算机
reboot
```

1.3.5 加入系统变量

在前面的内容中介绍了一些 Nginx 的操作命令，比如，要查看 Nginx 的版本信息，我们通常可以这样：

```
/usr/local/nginx/sbin/nginx -v
```

或

```
cd /usr/local/nginx/sbin
./nginx -v
```

使用这种方式，使得我们每次要执行相关命令时，都要输入很长的 Nginx 执行文件路径或者要先进入指定目录中才行，这确实有点儿麻烦。所以，我们可以将 Nginx 的路径配置到系统变量中。

1. 修改/etc/profile 文件

```
vim /etc/profile
```

2. 添加 PATH

在 profile 文件中添加 Nginx 执行文件的路径，如图 1-11 所示。

```
export ANDROID_NDK_ROOT=/opt/android-ndk-r14b
export ANDROID_SDK_ROOT=/opt/android-sdk-linux
export ANDROID_HOME=/opt/android-sdk-linux
export PATH=$ANDROID_NDK_ROOT:$ANDROID_SDK_ROOT/tools:$PATH
export ANT_HOME=/opt/apache-ant-1.9.9
export PATH=$ANT_HOME/bin:$PATH
export PATH=/usr/local/apr/bin:/usr/local/cmake-3.9.0/bin:$PATH
export PATH=$PATH:/usr/local/nginx/sbin
```

图 1-11

3. 使之立即生效

编辑/etc/profile 文件后，对于 PATH 的修改不会立马生效，如果要立即生效，则执行以下命令：

```
source /etc/profile
```

4. 执行 Nginx 命令

在环境变量生效之后，就可以直接用“nginx”来执行相关命令。比如，之前查看 Nginx 版本信息的命令，就可以直接执行：

```
nginx -v
```

1.4　Nginx 命令行

不同于其他软件系统，Nginx 仅有几个命令行参数，完全通过配置文件来进行配置。

在 1.3 节中介绍安装 Nginx 时，已经涉及了一些 Nginx 的基础命令行命令，比如显示 Nginx 的版本，启动和关闭 Nginx 等。本节会完整介绍 Nginx 相关的命令行命令。

1.4.1 命令行参数

如果已经配置了系统 PATH，则可以通过输入“nginx”来执行命令，否则要输入 Nginx 执行文件全路径。前面已经介绍了设置 PATH 的方法，所以下面就以“nginx”执行命令为例来介绍。

执行命令“nginx -?”，可以查看 Nginx 的命令帮助信息，执行结果如图 1-12 所示。

```
Usage: nginx [-?hvVtTq] [-s signal] [-c filename] [-p prefix] [-g directives]

Options:
  -?,-h         : this help
  -v            : show version and exit
  -V            : show version and configure options then exit
  -t            : test configuration and exit
  -T            : test configuration, dump it and exit
  -q            : suppress non-error messages during configuration testing
  -s signal     : send signal to a master process: stop, quit, reopen, reload
  -p prefix     : set prefix path (default: /usr/local/nginx/)
  -c filename   : set configuration file (default: conf/nginx.conf)
  -g directives : set global directives out of configuration file
```

图 1-12

1. nginx [-?hvVtTq]

nginx [-?hvVtTq]命令的介绍如下所示。

```
#查看帮助
nginx -?/-h
#显示版本信息
nginx -v
#显示版本和配置选项信息
nginx -V
#检测配置文件是否有语法错误
nginx -t
#测试配置文件，转储并退出
nginx -T
#检测配置文件时屏蔽非错误信息，只输出错误信息
nginx -q
```

2. nginx [-s signal]

在启动 Nginx 之后，可以通过使用-s 参数调用 Nginx 可执行文件来控制 Nginx。

```
#重新打开日志文件
nginx -s reopen
#快速停止 Nginx，此方式是先查出 Nginx 的主进程号，再使用 kill 命令强制“杀掉”进程
nginx -s stop
#优雅退出 Nginx（推荐使用，此方式会等待 Nginx 进程处理任务完毕再停止）
nginx -s quit
#重新加载配置并启动（平滑重启）
nginx -s reload
```

3．nginx [-c filename]

nginx [-c filename]命令用于在启动 Nginx 时指定配置文件，在实际应用时，filename 为配置文件全路径，例如：

```
nginx -c /usr/local/nginx/conf/nginx.conf
```

-c 表示 configuration，用于指定配置文件。如果不加-c 参数，则 Nginx 会默认加载其安装目录下的 conf 子目录中的 nginx.conf 文件。

4．nginx [-p prefix]

nginx [-p prefix]用于设置 Nginx 的前缀路径，默认值是/usr/local/nginx。

5．nginx [-g directives]

nginx [-g directives]是在配置文件之外设置全局命令。

```
nginx -g "pid /var/run/nginx.pid; worker_processes `sysctl -n hw.ncpu`;"
```

1.4.2　启动、停止和重启

1．启动 Nginx

要启动 Nginx，可运行其可执行文件。

Nginx 启动命令的格式：Nginx 执行文件-c Nginx 配置文件所在地址。

（1）用默认配置文件直接启动。

```
/usr/local/nginx/sbin/nginx
或
nginx
```

（2）指定配置文件启动。

```
nginx -c /usr/local/nginx/conf/nginx.conf
```

（3）其他方式启动。

```
service nginx start
```

2．停止 Nginx

停止 Nginx 的方法有 3 种：从容停止、快速停止和强制停止。一般都是通过发送系统信号给主进程的方式来停止 Nginx。

（1）查看 Nginx 的主进程号。

```
ps -ef | grep nginx
```

该命令在 1.3.3 节中介绍过，这里不再赘述。

（2）从容停止。

```
kill -QUIT 主进程号
```

或

```
kill -QUIT /usr/local/nginx/logs/nginx.pid
```

如果在 nginx.conf 配置文件中指定了 pid 文件存放的路径，则该文件中存放的就是 Nginx 当前的主进程号，其默认被放在 Nginx 安装目录的 logs 目录下。

（3）快速停止。

```
kill -TERM 主进程号
```

（4）强制停止。

```
pkill -9 nginx
```

（5）其他停止方式。

还可以用其他方式停止 Nginx 服务：

```
service nginx stop
```

或

```
nginx -s stop
```

3. 重启

如果修改了 Nginx 的配置文件，要想让新配置的文件生效，就得重启 Nginx。同样，可以发送系统信号给 Nginx 主进程来重启 Nginx。

（1）验证配置文件的正确性。

验证配置文件的正确性使用以下命令：

```
nginx -t
```

该命令会默认检查/usr/local/nginx/conf/nginx.conf 文件。如果要测试指定的配置文件，则执行以下命令（该命令中的文件路径要改成待测试的配置文件路径）：

```
nginx -t -c /usr/local/nginx/conf/nginx.conf
```

（2）发送信号重启 Nginx。

一旦主进程接收到重启 Nginx 的信号，它就会检查新配置文件语法的有效性，并尝试应用其中提供的配置。如果应用成功，则主进程启动新的工作进程，并将消息发送给旧的工作进程，请求关闭进程。否则，主进程回滚更改并继续使用旧的工作进程。旧的工作进程在接收到一个关闭命令后，停止接受新的连接，并继续服务当前的请求，直到服务完所有的请求。之后，旧

的工作进程退出。

```
kill -HUP 主进程号
```

或

```
nginx -s reload
```

（3）其他重启方式。

同样，还可以用其他方式来重启 Nginx：

```
service nginx reload
```

1.4.3　信号控制

可以通过向进程发送信号的方式来控制 Nginx。

在 1.4.2 节中，介绍了用系统信号来控制 Nginx 的停止和重启的方法。本节介绍一些常见的信号控制操作。

1．常见信号控制操作

主进程支持的信号如表 1-1 所示。

表 1-1

信　号	说　明
TERM，INT	快速关闭
QUIT	从容关闭，等请求结束后再关闭
HUP	改变配置文件后使用新配置启动新的工作进程，正常关闭旧的工作进程
USR1	重新打开日志文件，在切割日志时用途很大
USR2	平滑升级可执行程序
WINCH	从容关闭工作进程，并配合 USR2 信号来进行升级可执行程序

个别工作进程也可以通过信号来控制 Nginx，尽管这不是必需的。具体支持的信号如表 1-2 所示。

表 1-2

信　号	说　明
TERM，INT	快速关闭
QUIT	从容关闭，等请求结束后再关闭
USR1	重新打开日志文件
WINCH	调试异常停止（要求启用 debug_points）

2．具体语法

信号控制的具体语法为：Kill　-信号选项　Nginx 的主进程号。

例如：

```
Kill -INT 主进程号 #关闭 Nginx 进程
Kill -HUP 主进程号 #平滑地读取新配置文件，不必重启 Nginx
kill -USR1 主进程号
kill -USR2 主进程号
kill -WINCH 主进程号
```

1.4.4　平滑升级

当需要将正在运行的 Nginx 升级、添加/删除服务器模块时，如果先停止服务做相应的修改后再启动服务，则服务器在一段时间内会不能被访问。Nginx 的一大优势就是可以在不中断服务的情况下，使用新版本、重编译的 Nginx 可执行程序替换旧版本的可执行程序，这样就不会影响对服务器的访问。

下面介绍具体的升级步骤。

1．编译安装新的可执行程序

对于以编译源码方式安装的 Nginx，可以将新版本的 Nginx 编译安装到旧版本的安装路径中，从而用新版本的可执行程序替换旧版本的可执行程序。在替换之前，最好备份一下旧版本的可执行程序。

2．执行命令：kill -USR2 旧版本 Nginx 主进程号

执行该命令后，旧版本 Nginx 的主进程将它的 pid 文件重命名为 oldbin 文件（例如：/usr/local/nginx/logs/nginx.pid.oldbin），然后执行新版本的可执行程序，依次启动新版本的主进程和工作进程。此时，所有工作进程（包括旧版本和新版本）会同时运行，共同处理输入的请求。

3．执行命令：kill -WINCH 旧版本 Nginx 主进程号

如果要逐步停止旧版本的工作进程，则要发送 WINCH 信号给旧版本的主进程，然后，它的工作进程将被从容关闭。

一段时间后，旧版本的工作进程处理完所有已连接的请求后退出，仅由新版本的工作进程来处理输入的请求。

4. 恢复旧版本/使用新版本

这个时候，旧版本的主进程不会关闭其 listen sockets，并且可以管理它，以便在需要的时候重新启动它的工作进程。

如果新版本的可执行程序有问题，则可以恢复为旧版本，具体操作如下。

（1）kill -HUP 旧版本主进程号：Nginx 将在不重载配置文件的情况下，启动旧版本的工作进程。

（2）kill -QUIT 新版本主进程号：从容关闭新版本的工作进程。

（3）kill -TERM 新版本主进程号：强制退出新版本的工作进程。

（4）kill 新版本主进程号：如果因为某些原因新版本的工作进程不能退出，则向其发送 kill 信号。

如果新版本的主进程退出，则旧版本的主进程会被移除.oldbin 后缀，恢复为.pid 后缀，这样一切就恢复到版本升级之前了。

如果版本升级成功，而我们也希望保留新版本的服务器，则可以发送 QUIT 信号给旧版本的主进程，使其退出而只留下新版本的服务器运行。

1.5　Nginx 配置

Nginx 及其模块的工作方式是在配置文件中确定的。

Nginx 配置文件一般包括一个主配置文件和一些辅助的配置文件。这些配置文件均是纯文本文件，全部位于 Nginx 安装目录下的 conf 目录中，如图 1-13 所示。

```
-rw-r--r-- 1 root root 1077 Jan 22 15:15 fastcgi.conf
-rw-r--r-- 1 root root 1077 Jan 22 15:15 fastcgi.conf.default
-rw-r--r-- 1 root root 1007 Jan 22 15:15 fastcgi_params
-rw-r--r-- 1 root root 1007 Jan 22 15:15 fastcgi_params.default
-rw-r--r-- 1 root root 2837 Jan 22 15:15 koi-utf
-rw-r--r-- 1 root root 2223 Jan 22 15:15 koi-win
-rw-r--r-- 1 root root 3957 Jan 22 15:15 mime.types
-rw-r--r-- 1 root root 3957 Jan 22 15:15 mime.types.default
-rw-r--r-- 1 root root 2656 Jan 22 15:15 nginx.conf
-rw-r--r-- 1 root root 2656 Jan 22 15:15 nginx.conf.default
-rw-r--r-- 1 root root  636 Jan 22 15:15 scgi_params
-rw-r--r-- 1 root root  636 Jan 22 15:15 scgi_params.default
-rw-r--r-- 1 root root  664 Jan 22 15:15 uwsgi_params
-rw-r--r-- 1 root root  664 Jan 22 15:15 uwsgi_params.default
-rw-r--r-- 1 root root 3610 Jan 22 15:15 win-utf
```

图 1-13

在默认情况下，主配置文件名为 nginx.conf，也可以自定义主配置文件并加载自定义的配置文件启动 Nginx。由于除主配置文件 nginx.conf 外的文件都是在某些情况下才被使用的，只有主配置文件是在任何情况下都被使用的，所以，下面以主配置文件为例，介绍 Nginx 的配置。

1.5.1 配置命令

Nginx 包含由配置文件中指定的命令控制的模块。通常，nginx.conf 文件中的命令被分为简单命令和块命令。

1. 简单命令

一个简单的命令由名称和参数组成，并以分号结束。

命令名称是一个字符串，可以加单引号或双引号，也可以不加。但是如果名称中包含空格，则一定要加引号。

命令参数就是命令对应的配置值，使用空格或 Tab 字符与命令名称分隔。命令参数可以是一个或多个 Token 串，Token 串之间使用空格或者 Tab 字符分隔。例如：

```
error_log  logs/error.log  info;
```

其中“error_log”为命令名称，“logs/error.log”和“info”都为命令参数，该配置项指定了日志文件和错误日志级别。

2. 块命令

块命令与简单命令有着相同的结构，但不是以分号结束的，而是以一系列由大括号括起来的附加命令结束的。如果一个块命令在大括号内可以有其他的命令，那么它就被称为一个上下文（例如 events、http、server 和 location），例如：

```
location / {
    root   html;
    index  index.html index.htm;
}
```

3. 注释

配置文件中以“#”开始的行，或者前面有若干个空格或 Tab 字符，然后再以“#”结束的行，都被认为是注释。也就是说，注释只对编辑、查看文件的用户有意义，程序在读取这些注释行时，其实际的内容是被忽略的。例如：

```
#全局错误日志
#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;
```

1.5.2 配置上下文

在 1.5.1 节介绍块命令时提到了上下文的概念。

nginx.conf 文件中的配置信息，根据其逻辑上的意义被分成了多个作用域，不同的作用域分别用大括号来限定。这些大括号限定的区域被称为上下文，其中容纳着相关配置细节。

不同的作用域含有一个或者多个配置项，这些区域负责提供组织化结构，用于决定是否应用其中包含的配置。

Nginx 的几个主要命令的上下文介绍如表 1-3 所示。

表 1-3

上下文	说　明
main	配置影响 Nginx 全局的命令，Nginx 运行时与具体业务功能（如 HTTP 服务或 E-mail 服务代理）无关的一些参数。如工作进程数、运行的身份等
http	提供与 HTTP 服务相关的一些配置参数。如是否使用 keep-alive，是否使用 GZIP 进行压缩等。可以嵌套多个 server、配置代理、缓存、日志定义等绝大多数功能和第三方模块的配置
server	HTTP 服务上支持若干台虚拟主机，每个虚拟主机对应一个 server 配置项，配置项里面包含该虚拟主机相关的配置。在提供 E-mail 服务代理时，也可以建立若干个 server，每个 server 通过监听的地址来区分
location	在 HTTP 服务中，某些特定的 URL 对应的一系列配置项
mail	实现 E-mail 服务相关的 SMTP/IMAP/POP3 代理时共享的一些配置项（因为可能实现多个代理，工作在多个监听地址上）
events	配置影响 Nginx 服务器或与用户的网络连接，包括每个进程的最大连接数，选取哪种事件驱动模型处理连接请求，是否允许同时接受多个网络连接，开启多个网络连接序列化等
upstream	配置 HTTP 负载均衡器分配流量到几个应用程序服务器

上下文可能会出现包含的情况，通常 http 和 mail 一定在 main 中，server 在 http 中，location 在 server 中。在一个上下文里，可能包含多个其他类型的上下文。例如，如果 HTTP 服务支持了多个虚拟主机，那么在 http 里，就会出现多个 server。命令只能作用于其设计所面向的对应上下文内。Nginx 会将超出指定上下文的命令视为错误。在 Nginx 官网中详细介绍了每个命令适用的上下文。

1.5.3　配置文件结构

nginx.conf 文件的组织方式采用树状结构。

```
... #main 全局块
events { #events 块
  ...
}

http{  #http 块

  ... #http 全局块
```

```
  upstream … #upstream 负载均衡块
  {
    ...
  }

  server {  #server 块

    ... #server 全局块

    location [PATTERN] #location 块
    {
      ...
    }
  }

  server
  {
    ...
  }

  ... #http 全局块

}

mail{ #mail 块
  ...
}
```

1.5.4 配置变量

Nginx 配置文件支持使用变量，可以使用内置变量，也可以自定义变量。

1. 内置变量

Nginx 可以使用内置变量来配置命令。例如，配置日志格式如下：

```
log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                  '$status $body_bytes_sent "$http_referer" '
                  '"$http_user_agent""$http_x_forwarded_for"';

$remote_addr 与$http_x_forwarded_for：用来记录客户端的 IP 地址
$remote_user：用来记录客户端的用户名称
$time_local：  用来记录访问时间与时区
$request：  用来记录请求的 URl 与 HTTP 协议
$status：  用来记录请求状态，成功是 200
$body_bytes_sent ：用来记录发送给客户端文件主体内容大小
```

```
$http_referer：用来记录客户从哪个页面链接访问过来的
$http_user_agent：用来记录客户浏览器的相关信息
```

Nginx 变量名前面有一个“$”符号，这是语法上的要求。在配置文件中引用变量时都必须加上“$”前缀。http 核心模块的内置变量可参考 Nginx 官网。

2. 自定义变量

用户也可以自定义变量，语法为：set var_name value。例如：

```
set $a "hello world";
```

这里使用了标准 ngx_rewrite 模块的 set 配置命令对变量“$a”进行赋值操作，把字符串“hello world”赋给了它。虽然添加“$”这样的变量前缀会让 Java 和 C#程序员感到不舒服，但这种表示方法的好处也是显而易见的，那就是可以直接把变量嵌入字符串常量中，以构造出新的字符串。例如：

```
set $a "hello world";
set $b "$a, $a";
```

这里通过已有的 Nginx 变量的值来构造变量 b 的值，当这两条命令顺序执行完之后，$a 的值是“hello world”，而$b 的值则是“hello world，hello world”。

1.5.5　配置实例

前面介绍了配置文件的相关知识点，本节会举一个完整的实例，介绍一些简单的 Nginx 配置命令。实例如下：

```
#运行用户
#user  nobody;
#工作进程数，通常设置成和 CPU 的数量相等或是其两倍
worker_processes  1;

#全局错误日志
#error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;

#PID 文件
#pid        logs/nginx.pid;

#工作模式及连接数上限
events {
#单个后台工作进程的最大并发连接数
    worker_connections  1024;
}
```

```
#设定 HTTP 服务器
http {
    #设定 mime 类型，类型由 mime.type 文件定义
    include       mime.types;
    #默认文件类型
    default_type  application/octet-stream;
    #设定日志格式，$remote_addr 这些以"$"开头的变量是内置变量
    #log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
    #                  '$status $body_bytes_sent "$http_referer" '
    #                  '"$http_user_agent""$http_x_forwarded_for"';

    #access_log  logs/access.log  main;

    #sendfile 命令指定 Nginx 是否调用 sendfile 函数（zero copy 方式）来输出文件
    #对于普通应用程序，必须设为 on
    #以平衡磁盘与网络 I/O 处理速度，减少系统的升级时间
    sendfile        on;
    #tcp_nopush     on;

    #连接超时时间
    #keepalive_timeout  0;
    keepalive_timeout  65;

    #开启 gzip 压缩
    #gzip  on;

    #设定虚拟主机配置
    server {
#监听 80 端口
        listen       80;
#定义使用 localhost 访问
        server_name  localhost;

        #charset koi8-r;
#设定本虚拟主机的访问日志
        #access_log  logs/host.access.log  main;
#默认请求
        location / {
#定义默认网站根目录位置
            root   html;
#定义首页索引文件的名称
            index  index.html index.htm;
        }

        #error_page  404              /404.html;

        # 定义错误提示页面
```

```
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
}
```

在该实例中，涉及的配置区域有：main、events、http、server 和 location。更多配置命令可参考 Nginx 官方文档。

1.6　Nginx 模块化体系

Nginx 具有 Web 服务器的基础功能，同时具有 Web 服务反向代理及 E-mail 服务反向代理功能。

Nginx 的内部结构是由核心部分和一系列的功能模块所组成的。这样划分是为了使每个模块的功能相对简单，便于开发，同时也便于对系统进行功能扩展。为了便于描述，在下文中使用 Nginx Core 来表示 Nginx 的核心功能部分。

1.6.1　模块概述

Nginx 将各功能模块组织成一个链条，当有请求到达时，请求依次经过链条上的部分或者全部模块，然后进行处理。每个模块实现特定的功能。例如，实现对请求解压缩的模块，实现 SSI 的模块，实现与上游服务器进行通信的模块，实现与 FastCGI 服务进行通信的模块。各模块之间的关系如图 1-14 所示。

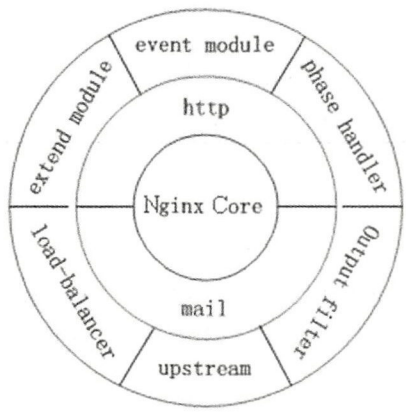


图 1-14

Nginx Core 实现了底层的通信协议，为其他模块和 Nginx 进程构建了基本的运行环境，并

且构建了其他各模块协作的基础。

http 模块和 mail 模块位于 Nginx Core 与各功能模块的中间，这两个模块分别处理与 HTTP 协议和 E-mail 协议（SMTP/IMAP/POP3）有关的事件，并且确保这些事件能以正确的顺序调用其他的功能模块。目前，HTTP 协议是在 http 模块中实现的，但是有可能在将来会被独立到一个单独的模块中，以扩展 Nginx 支持的 SPDY 协议。

除此之外，大部分与协议相关的或者与应用程序相关的功能都是通过这些模块实现的。

1.6.2 模块分类

Nginx 的模块根据其功能可以分为如表 1-4 所示的 6 种类型。

表 1-4

模块类型	说　明
event module	搭建了独立于操作系统的事件处理机制的框架，以及提供了各具体事件的处理模块，包括 ngx_events_module，ngx_event_core_module 和 ngx_epoll_module 等。Nginx 具体使用何种事件处理模块，依赖于具体的操作系统和编译选项
phase handler	此类型的模块也被称为 handler 模块。主要负责处理客户端请求并产生待响应内容，比如 ngx_http_static_module 模块，负责客户端的静态页面请求处理并将对应的磁盘文件准备为响应内容输出
output filter	也被称为 filter 模块，主要负责对输出的内容进行处理，可以对输出进行修改。例如，可以实现对输出的所有 HTML 页面增加预定义的 Footbar 一类的工作，或者对输出的图片的 URL 进行替换之类的工作
upstream	实现反向代理的功能，将真正的请求转发到后端服务器中，并从后端服务器中读取响应及发回客户端。upstream 模块是一种特殊的 handler 模块，只不过其响应内容不是真正由自己产生的，而是从后端服务器上读取的
load-balancer	实现特定的算法，在众多的后端服务器中，选择一个服务器作为某个请求的转发服务器
extend module	根据特定业务需要编写的第三方模块

1.7 本章小结

本章介绍了 Nginx，包括它的应用、优点、实际的安装和配置，以及 Nginx 模块的概念。在对 Nginx 有所了解的基础上，我们可以进一步地探究、了解 Nginx-rtmp-module。

第2章

Nginx-rtmp-module 基础

随着移动网络的普及，4G、5G 网络的接入，在各大移动通信服务提供商不断推出免流量、大流量通信服务套餐的环境下，直播行业更是加快了发展的脚步，快速融入了人们的生活、娱乐、学习中。各大厂家推出的直播平台，更是包含了一整套的服务器体系，这其中有什么奥秘呢？本章为读者打开直播技术的大门。

第 1 章介绍了 Nginx，从本章开始介绍 Nginx-rtmp-module（以下简称 NRM）的相关知识。虽然，现在在市场上有很多第三方直播平台，但是作为开发者，选择一个入手简单、功能强大的 NRM 作为切入点学习搭建直播系统也是很不错的选择，下面会一步步地介绍直播系统的搭建。最终让读者能够自己搭建一个简单的直播服务系统是本书的目的。

2.1 Nginx-rtmp-module 介绍

NRM 的出现使得非专业流媒体开发工程师也可以简单、迅速地搭建流媒体服务器。

NRM 的应用特性包含以下几种。

- 支持 RTMP、HLS、MPEG-DASH 直播。
- 支持 RTMP、HLS 点播。
- 可以将一次直播分为多个视频文件存储。
- 支持 H.264 视频编/解码或 AAC 音频编/解码。
- 支持 FFmpeg 命令内嵌。
- 支持回调 HTTP。

- 可以使用 HTTP 对直播进行控制，如删除/录播。
- 具有更优秀的缓存技术，确保在效率与解码之间达到平衡，获得更好的效果。
- 支持更多的操作系统，如 Linux、FreeBSD、MacOS、Windows。

2.2 RTMP 协议与 HLS 协议

Real Time Messaging Protocol，实时消息传送协议（RTMP 协议），它是 Adobe 公司为 Flash 播放器和服务器之间传输音/视频和数据而开发的私有协议。

HTTP Live Streaming(HLS)是苹果公司的开发标准。它最初是由苹果公司针对 iPhone、iPod、iTouch 和 iPad 等移动设备而开发的流。由于 HLS 协议是基于 HTTP 的，因此，其继承了很多 HTTP 的优点。

2.2.1 RTMP 协议

RTMP 协议是为了在 Adobe Flash 平台技术（包括 Adobe Flash Player 和 Adobe AIR）之间高性能传输音/视频和数据而设计的。RTMP 协议是一个开放的规范，通过此规范可以创建产品和技术，可以通过 OpenAMF、SWF、FLV 和 F4V 格式，提供与 Adobe Flash Player 兼容的视/音频和数据。

1. 关键特性

RTMP 协议是应用层协议，需要靠底层可靠的传输层协议（通常是 TCP）来保证信息传输的可靠性。在建立完基于传输层协议的链接后，RTMP 协议也需要客户端和服务器端通过“握手”来建立基于传输层连接之上的 RTMP 连接。在连接上会传输一些控制信息，如 SetChunkSize 和 SetACKWindowSize。其中 CreateStream 命令会创建一个 Stream 链接，用于传输具体的音/视频等数据，以及控制这些信息传输的命令信息。RTMP 协议在传输时会对数据进行格式化，这种被格式化的消息被称为 RTMP Message。而在实际传输时，为了更好地实现多路复用、分包和信息的公平性，发送端会把 Message 划分为带有 Message ID 的 Chunk。每个 Chunk 可能是一个单独的 Message，也可能是 Message 的一部分，在接收端会根据 chunk 中包含的 data 的长度、message id 和 message 的长度，把 chunk 还原成完整的 Message，从而实现信息的收发。

2. 握手

一个 RTMP 连接以“握手”开始。这里的“握手”和其他协议的“握手”不一样。这里的“握手”由 3 个固定大小的 chunk 组成，而不是由可变大小的带有头文件的 chunk 组成。

客户端（发起连接的一方）和服务器端各自发送 3 个相同的块。这些块如果是客户端发送的，则记为 C0、C1 和 C2，如果是服务器端发送的，则记为 S0、S1 和 S2，如图 2-1 所示。

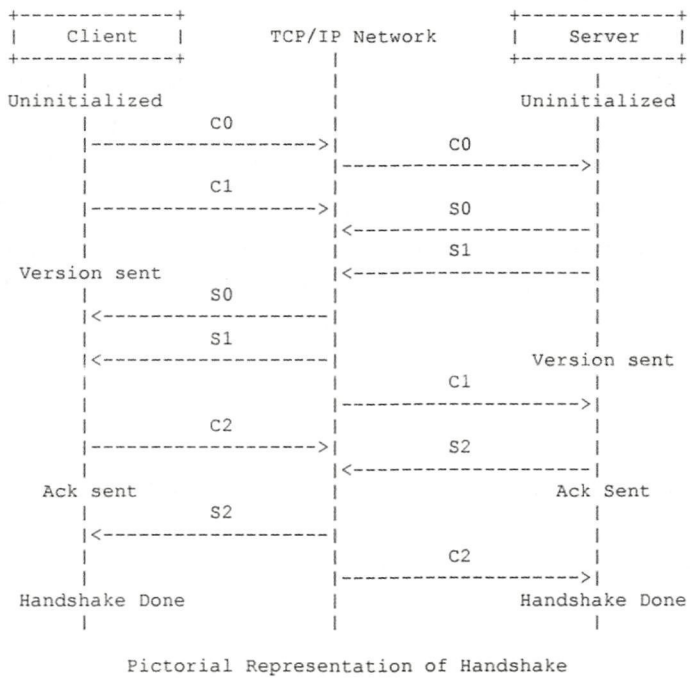


图 2-1

3. 过程

- 客户端要等收到 S1 之后才能发送 C2。
- 客户端要等收到 S2 之后才能发送其他信息（控制信息和真实音/视频等数据）。
- 服务器端要等收到 C0 之后才能发送 S1。
- 服务器端必须要等收到 C1 之后才能发送 S2。
- 服务器端必须要等收到 C2 之后才能发送其他信息（控制信息和真实音/视频等数据）。

2.2.2　HLS 协议

HLS 用于将实时和按需的音/视频内容流到 iPhone、iPad、iPod Touch、Apple TV 和 Mac 中。HTTP 允许用户使用普通的 Web 服务器（而不是专门的流媒体服务器）轻松地将媒体内容部署到流中。HLS 流的行为类似于常规的 Web 流量。它们使用现有的缓存基础设施，如内容传递网

络（CDNS），并可靠地通过典型的防火墙和路由器。HLS 可以适应可变的网络条件，动态地调整回放，以匹配有线和无线连接的可用速度。

除可靠和易于部署外，HLS 还支持的功能有：关闭字幕、快速转发和反向播放、备用音/视频、插入广告，以及保护内容。

在典型的 HLS 流中，支持 HLS 的视频编码器解决方案接收一个实时视频提要或分发的媒体文件。编码器在不同的比特率、分辨率和质量级别上创建了多个版本（称为变体）的音/视频。然后编码器将这些变体分割成一系列的小文件，它们被称为媒体段。与此同时，编码器为每个变量创建一个媒体播放列表文件，其中包含指向该变体的媒体段的 URL 列表。编码器还创建了一个主播放列表文件，其中包含对可变媒体播放列表的 URL 列表，以及控制流播放行为的描述性标记。在生成播放列表和片段时，编码器或自动脚本将文件上载到 Web 服务器或 CDN 中。通过在 Web 页面中嵌入主播放列表文件的链接，或者创建自己的自定义应用程序来下载主播放列表文件，用户可以提供对内容的访问。

1. 关键特性

HLS 技术允许用户通过 HTTP 传输内容，并允许在网络环境变化时自动切换流。这些属性使得 HLS 成为一个很好的媒体发布的解决方案。此外，该技术还包括可用性、可用性和安全性等特性。用户可以使用以下可扩展的解决方案来保证质量。

2. M3U8

编码器将媒体播放列表保存在 M3U 格式的文本文件（m3u8 文件）中。媒体播放列表中包含媒体段的 URL 和播放所需的其他信息。播放列表的三种类型：实时、事件或视频点播（VOD），决定了如何导航流。

实时播放列表让观众在有限的时间范围内可以快速地前进和反向播放。在活动结束之前，该程序的时间范围会被一直向前推进。

事件播放列表让观众返回到流的开始。

VOD 播放列表表示一个以前完成的程序，可以从头到尾完全导航。

当新创建的媒体段在服务器上可用时，实时和事件类型的程序都需要更新媒体播放列表。编码器将新媒体段引用并添加到播放列表的末尾，并将更新后的播放列表上传至服务器中。在 live 播放列表中，可以从媒体播放列表中删除对旧媒体段的引用，并将一个滑动窗口提供给连续流。

（1）一个简单的 m3u8 文件。

```
#EXTM3U
#EXT-X-VERSION:6
```

```
#EXT-X-TARGETDURATION:10
#EXT-X-MEDIA-SEQUENCE:26
#EXTINF:9.901,
http://xx.easy.com/a/segment1.ts
#EXTINF:9.901,
http://xx.easy.com/a/segment2.ts
#EXTINF:9.501,
http://xx.easy.com/a/segment3.ts
```

播放列表中的#EXT-X-PLAYLIST-TYPE:EVENT 标签名称告诉媒体播放器，这个播放列表的行为将与一个实时媒体播放列表不同。事件播放列表保留对旧媒体的引用，同时获得新的引用。在这个过程中会产生一个扩展的媒体播放列表。这种类型的播放列表允许观众从程序开始时自由地（向后或向前）导航。由于所有媒体片段的引用都保留在活动列表的播放列表中，所以，事件播放列表很容易被转换为 VOD 播放列表。

（2）一个媒体播放列表。

```
#EXTM3U
#EXT-X-VERSION:6
#EXT-X-TARGETDURATION:10
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-PLAYLIST-TYPE:EVENT <-----
#EXTINF:9.901,
http://xx.easy.com/a/segment1.ts
#EXTINF:9.901,
http://xx.easy.com/a/segment2.ts
#EXTINF:9.501,
http://xx.easy.com/a/segment3.ts
```

VOD 播放列表包含了对所有可用媒体段的引用，这种播放列表允许观众浏览整个程序。#EXT-X-ENDLIST 标签标志着可下载媒体片段的结束。

（3）一个完整的 VOD 播放列表。

```
#EXTM3U
#EXT-X-VERSION:6
#EXT-X-TARGETDURATION:10
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-PLAYLIST-TYPE:VOD
#EXTINF:9.9001,
http://media.example.com/wifi/segment0.ts
#EXTINF:9.9001,
http://media.example.com/wifi/segment1.ts
#EXTINF:9.9001,
http://media.example.com/wifi/segment2.ts
#EXT-X-ENDLIST
```

3. 主播放列表

主播放列表为流中的每个媒体播放列表提供一个地址。主播放列表中还包括了重要的细节数据，如带宽、分辨率和编解码器。观众通过这些信息可以决定最合适设备的变体和当前测量的可用带宽。

主播放列表中显示了 4 个变体。在主播放列表中，媒体播放列表的顺序无关紧要，除非你启动了流。之后观众开始下载主播放列表可以播放的第一个版本。如果条件允许，则观众可切换到另一个媒体播放列表中的流。

下面显示了播放列表与切片关系。

```
    #EXTM3U
    #EXT-X-VERSION:6
    #EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=2855600,CODECS="avc1.4d001f,mp4a.40.2",R
ESOLUTION=960x540
    live/medium.m3u8
    #EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=5605600,CODECS="avc1.640028,mp4a.40.2",R
ESOLUTION=1280x720
    live/high.m3u8
    #EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=1755600,CODECS="avc1.42001f,mp4a.40.2",R
ESOLUTION=640x360
    live/low.m3u8
    #EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=545600,CODECS="avc1.42001e,mp4a.40.2",RE
SOLUTION=416x234
    live/cellular.m3u8
```

观众只能下载一次主播放列表。然而，媒体播放列表的下载次数因播放列表类型而异。对于实时广播和事件广播，播放器在每个媒体段持续时间后下载媒体播放列表文件，因为播放列表可能随着新的媒体段的更新或随着流的进度而丢失旧的媒体段。对于 VOD，观众只能下载一次媒体播放列表，如图 2-2 所示。

4. 快速向前和反向播放

HLS 通过使用 I-frame 播放列表，支持快速向前和反向回放。I-frame 播放列表指向已经存在的媒体段内的一个字节范围。快进和反向回放不需要特殊的媒体段。

5. 交替音频和视频播放

HLS 主播放列表提供了多种音频渲染（这是本地化的一个很有价值的特性），例如，你的主播放列表中可能包括多种语言音轨，如法语、德语、西班牙语和英语，音轨中包含未被屏蔽（信号未被组合）的音频片段。HLS 还支持多个视频流，例如，体育赛事的多个摄像机视频分屏等。

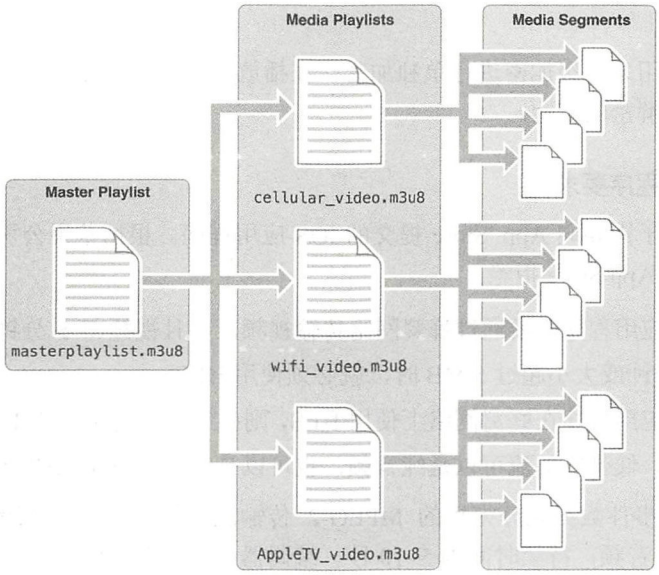


图 2-2

6．回退和流交替

在主播放列表中，替代媒体播放列表可以根据带宽或设备的情况作为替代品。如果播放器不能重新加载媒体播放列表文件——由于诸如 404 错误、服务器崩溃或内容分发服务器存在节点问题，则会试图切换到另一个服务器上提供的兼容媒体播放列表。HLS 提供具有相同带宽的多个媒体播放列表，播放器切换为相同的播放列表，提供一致的流性能。

7．定时的元数据

用户可以向媒体段添加各种元数据。这些数据在回放期间为应用程序提供了额外的信息。例如，将艺术家的名字和歌曲的标题添加到音频流中，或者将当前目标人物的名字和统计数字添加到视频流中。

在给定的时间偏移量中插入额外的数据，被称为定时元数据（其为可选的，在给定的时间之后将定时元数据插入到所有片段中）。

8．广告插入

HLS 通过间断标记，可以将广告、播放列表中的标记平滑地在不同内容之间转换。

9. 内容保护

媒体段可以使用采样级加密进行单独加密。在播放列表文件中显示了相应的关键文件，这样用户就可以获取解密的密钥。

10. iOS 应用程序要求

以下要求适用于需要向 App Store 提交的 iOS 应用程序。根据苹果公司的规定，不兼容的应用程序可能会被 App Store 拒绝。

- 如果你的应用程序需要通过蜂窝网络传输视频，并且视频在 5 分钟内，则当视频时长超过 10 分钟或大小超过 5 MB 时，就必须使用 HLS。
- 如果你的应用程序在蜂窝网络上使用 HLS，则必须至少提供一个 192 Kbps 或具有更低带宽的流。低带宽的流可以是纯音频，也可以是带有静态图像的音频。

编码器通过将事件数据划分为短的 MPEG-2 传输流文件来创建媒体段。通常，文件包含 H.264 视频或 AAC 音频，持续时间为 5~10 秒。编码器允许设置媒体段的编码和持续时间。

2.3 NRM 的搭建

首先，请先参照第 1 章，在 Linux 系统中成功安装 Nginx。这里所使用的版本为 CentOS，使用工具为 Xshell。

1. 下载 NRM

在 GitHub 中打开 Nginx-rtmp-module 首页，在其中选择“Download.tar.gz”选项，如图 2-3 所示。

图 2-3

这里所下载的版本为 arut-nginx-rtmp-module-v1.2.1-0-g791b613.tar.gz。当然也可以使用 wget 命令下载。wget 是一个从网络上自动下载文件的工具，支持通过 HTTP、HTTPS、FTP 这

3 个最常见的 TCP/IP 协议下载，并可以使用 HTTP 代理（wget 是 World Wide Web 与“get”的结合）。

```
    yum install -y wget
    wget -O "NRM.tar.gz"
https://codeload.github.com/arut/nginx-rtmp-module/legacy.tar.gz/master
```

2. 安装 lrzsz 程序

lrzsz 是一款在 Linux 里可代替 FTP 上传和下载的程序。

```
    yum install -y lrzsz
```

3. 创建一个目录

```
    mkdir /download
```

4. 上传 tar 文件

输入 rz 命令并按 Enter 键，然后选择要下载的 arut-nginx-rtmp-module-v1.2.1-0-g791b613.tar.gz 文件。

5. 修改文件名

用户可以使用 mv 命令为文件或目录重命名，或者将文件由一个目录移入另一个目录中。该命令如同 MS-DOS 下的 ren 和 move 命令的组合。rename 是一个可以对多个文件重命名的 mv 命令。

```
    mv arut-nginx-rtmp-module-v1.2.1-0-g791b613.tar.gz NRM.tar.gz
```

也可以使用 rename 命令 。

```
    rename arut-nginx-rtmp-module-v1.2.1-0-g791b613 NRM
arut-nginx-rtmp-module-v1.2.1-0-g791b613.tar.gz
```

6. 解压缩文件

tar 是 Linux 中用来解压缩文件的命令。

```
    tar -xvzf NRM.tar.gz
```

7. 简化目录

简化目录是为了更方便地输入后续的操作命令，简化目录后的结果如图 2-4 所示。

图 2-4

```
rename arut-nginx-rtmp-module-791b613 NRM arut-nginx-rtmp-module-791b613
```

8. 检查 Nginx 配置

```
./nginx -V # 注意 V 要大写，如图 2-5 所示
```

在“./nginx –v”中，输入小写“v”只会显示版本信息。

```
[root@NRM nginx-1.12.2]# pwd
/usr/local/nginx-1.12.2
[root@NRM nginx-1.12.2]# cd sbin/
[root@NRM sbin]# ./nginx -V
nginx version: nginx/1.12.2
built by gcc 4.4.7 20120313 (Red Hat 4.4.7-18) (GCC)
configure arguments: --prefix=/usr/local/nginx
```

图 2-5

9. 配置 NRM 到 Nginx 中

加载 NRM 库，并输出到指定目录中，最后开启 debuglog，如图 2-6 所示。

```
./configure --add-module=/download/NRM --prefix=/usr/local/nginx --with-debug
make & make install
```

```
[root@NRM sbin]#./nginx -V
nginx version: nginx/1.12.2
built by gcc 4.4.7 20120313 (Red Hat 4.4.7-18) (GCC)
built with OpenSSL 1.0.1e-fips 11 Feb 2013
TLS SNI support enabled.
configure arguments: --add-module=/download/NRM --prefix=/usr/local/nginx --with-debug
```

图 2-6

10. 启动 Nginx

```
./nginx
```

到这里 NRM 组件安装完毕。

2.4 搭建第一个直播系统

在 2.4 节介绍了搭建 NRM 环境，下面搭建一个简单的直播系统。

1. 进入配置文件目录

打开配置文件目录：

```
cd /usr/local/nginx/conf/
```

其中 nginx.conf 为默认加载的配置文件，nginx.conf.default 为默认配置的文件备份，如图 2-7 所示。

```
[root@NRM conf]# ll
total 60
-rw-r--r-- 1 root root 1077 Jan 22 14:25 fastcgi.conf
-rw-r--r-- 1 root root 1077 Jan 22 14:25 fastcgi.conf.default
-rw-r--r-- 1 root root 1007 Jan 22 14:25 fastcgi_params
-rw-r--r-- 1 root root 1007 Jan 22 14:25 fastcgi_params.default
-rw-r--r-- 1 root root 2837 Jan 22 14:25 koi-utf
-rw-r--r-- 1 root root 2223 Jan 22 14:25 koi-win
-rw-r--r-- 1 root root 3957 Jan 22 14:25 mime.types
-rw-r--r-- 1 root root 3957 Jan 22 14:25 mime.types.default
-rw-r--r-- 1 root root 2656 Jan 22 14:25 nginx.conf
-rw-r--r-- 1 root root 2656 Jan 22 14:25 nginx.conf.default
-rw-r--r-- 1 root root  636 Jan 22 14:25 scgi_params
-rw-r--r-- 1 root root  636 Jan 22 14:25 scgi_params.default
-rw-r--r-- 1 root root  664 Jan 22 14:25 uwsgi_params
-rw-r--r-- 1 root root  664 Jan 22 14:25 uwsgi_params.default
-rw-r--r-- 1 root root 3610 Jan 22 14:25 win-utf
```

图 2-7

2. 修改默认配置文件

修改系统服务脚本 init.d：

```
cd /etc/init.d/
vim nginx
```

将 nginx.conf 修改为 live.conf：修改默认 service nginx 服务指向的配置文件，如图 2-8 所示。

```
[                =       ] && exit 0

nginx=
prog=$(basename $nginx)

NGINX_CONF_FILE=

[ -f /etc/sysconfig/nginx ] && . /etc/sysconfig/nginx

lockfile=/var/lock/subsys/nginx

start() {
    [ -x $nginx ] || exit 5
    [ -f $NGINX_CONF_FILE ] || exit 6
    echo -n $
    daemon $nginx -c $NGINX_CONF_FILE
    retval=$?
    echo
    [ $retval -eq 0 ] && touch $lockfile
    return $retval
}
```

图 2-8

```
NGINX_CONF_FILE="/usr/local/nginx/conf/live.conf"
rename nginx live nginx.conf
#重命名 Nginx 为 live 目标文件 nginx.conf
```

3. 精简及确认 live.conf

精简后的 live.conf 中去除了默认注释：

```
worker_processes  1;
events {
    worker_connections  1024;
}
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    server {
        listen       80;
        server_name  localhost;
        location / {
            root   html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
}
#修改文件名后，我们将系统产生的配置文件进行精简并检查结果
service nginx configtest
#nginx: the configuration file /usr/local/nginx/conf/live.conf syntax is ok
#nginx: configuration file /usr/local/nginx/conf/live.conf test is successful
```

4. 配置 RTMP 直播

（1）RTMP 标签。

NRM 的基础标签的所有服务都被配置在 RTMP 标签中。

（2）Server 标签。

Server 标签是服务标签，一个 RTMP 服务中可以有多个 Server 标签，每个 Server 标签可以监听不同端口，Server 标签中的配置是应用于所有 Application 标签的。

（3）Application 标签。

Application 标签是应用标签，一个 Server 标签中可以有多个 Application 标签，Application 标签中的配置是应用于其本身的，application name 确保了在请求时进行准确的 Application 划分。

```
worker_processes  1;
events {
    worker_connections  1024;
}
rtmp{
```

```
    server{
        listen 1935;
        application mylive{
            live on;
        }
    }
}
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    server {
        listen       80;
        server_name  localhost;
        location / {
            root   html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
}
```

5. 防火墙规则

```
vim /etc/sysconfig/iptables
#在 iptables 中可以设置防火墙规则
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -p icmp -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 21 -j ACCEPT
-A INPUT -p tcp --dport 1935 -j ACCEPT
#开启 TCP 的 1935 端口写入权限,-p 代表了指定协议
-A INPUT -j REJECT --reject-with icmp-host-prohibited
-A FORWARD -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

RTMP 协议是基于 TCP 协议的。重启防火墙使配置生效。

```
service iptables restart
```

6. 推流

FFmpeg 与推/拉流会在第 3 章中说明，如图 2-9 所示为推流中的 FFmpeg。

```
ffmpeg -i f:\xx.flv -r 25 -b 4M -f flv rtmp://172.26.22.30:1935/mylive/6
```

```
    Stream #0:0: Video: flv1 (flv) ([2][0][0][0] / 0x0002), yuv420p, 1280x720, q=2-31, 200 kb/s,
fps, 1k tbn, 30 tbc
    Metadata:
      encoder         : Lavc57.107.100 flv
    Side data:
      cpb: bitrate max/min/avg: 0/0/200000 buffer size: 0 vbv_delay: -1
    Stream #0:1: Audio: mp3 (libmp3lame) ([2][0][0][0] / 0x0002), 22050 Hz, stereo, s16p
    Metadata:
      encoder         : Lavc57.107.100 libmp3lame
frame= 3013 fps= 67 q=31.0 size=   21173kB time=00:02:05.66 bitrate=1380.2kbits/s speed= 2.8x
```

图 2-9

7. 拉流

这里使用 VLC 流媒体播放器来拉流，此时正在直播的动画片可以观看了，如图 2-10 所示。

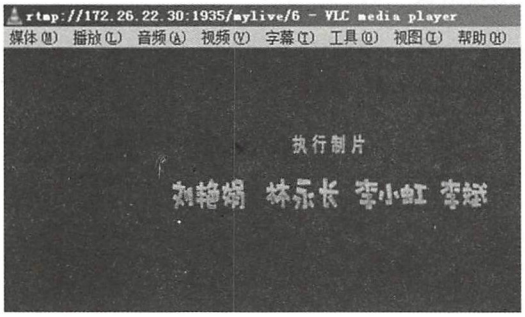


图 2-10

2.5 本章小结

本章内容主要包括：

- RTMP：Adobe 面向 Flash 的流媒体协议。
- HLS：苹果公司面向苹果产品的流媒体协议。
- Nginx-rtmp-module：Nginx 中的./configure --add-module。

本章介绍了 RTMP、HLS 两种协议，并介绍了如何安装及搭建 Nginx-rtmp-module 环境，最后还完成了第一个直播系统。

第3章

Nginx-rtmp-module 进阶

在第 2 章中简单地介绍了 RTMP 与 HLS 所对应的不同平台，以及如何成功搭建第一个直播系统。本章会介绍如何配置一个基于 HLS 协议的直播系统，以及 m3u8 与 ts 文件的配置，然后逐渐深入介绍 NRM 与直播系统的高级应用。

3.1 如何使 NRM 支持 HLS 协议直播

在 2.5 节中我们搭建了一个基于 RTMP 协议的直播系统，那么基于 HLS 协议的直播系统是如何搭建的？在 HLS 协议直播系统中，m3u8 和 ts 文件被存放在哪里？基于 HLS 协议的直播系统是否可以与基于 RTMP 协议的直播系统一起存在？这些在本节会进行介绍。

1. 为 m3u8 文件创建一个预备目录

```
mkdir /usr/local/m3u8File
```

2. 修改配置文件

```
application mylive{
          live on;#开启直播
          hls  on;#开启 HLS 直播
          hls_path /usr/local/m3u8File;
          #配置 HLS m3u8 文件存放地址

   }
```

因为HLS协议是基于HTTP协议的，所以我们无法通过RTMP协议头访问HLS m3u8文件，因此，下面在http标签下配置它的访问操作。

```
    http{
       server{
           listen 80;
           location /mylive_hls{
                       types {
                           #m3u8 type 设置
                           application/vnd.apple.mpegurl m3u8;
                           #ts 分片文件设置
                           video/mp2t ts;
                       }
         #指向访问 m3u8 文件目录
          alias /usr/local/m3u8File;
                add_header Cache-Control no-cache;#禁止缓存

                }
          }
    }
```

3. 推流

执行推流命令，这里额外配置了视频编码器 libx264 与音频编码器 aac：

```
    ffmpeg -i /tmp/nh.mp4 -vcodec libx264 -acodec aac -f flv
rtmp://172.26.22.30:1935/mylive/44
```

通过前面的配置，我们可以得知 m3u8 和 ts 文件被存放在/usr/local/m3u8File 目录下，查看这个目录可以得知是否有 m3u8 及 ts 文件输出，如图 3-1 所示。

```
[root@NRM m3u8File]# ll -h
total 25M
-rw-r--r-- 1 nobody nobody 2.2M Jan 25 08:51 44-20.ts
-rw-r--r-- 1 nobody nobody 1.3M Jan 25 08:51 44-21.ts
-rw-r--r-- 1 nobody nobody 1.5M Jan 25 08:51 44-22.ts
-rw-r--r-- 1 nobody nobody 6.0M Jan 25 08:51 44-23.ts
-rw-r--r-- 1 nobody nobody 6.3M Jan 25 08:51 44-24.ts
-rw-r--r-- 1 nobody nobody 1.1M Jan 25 08:51 44-25.ts
-rw-r--r-- 1 nobody nobody 3.0M Jan 25 08:51 44-26.ts
-rw-r--r-- 1 nobody nobody 2.4M Jan 25 08:51 44-27.ts
-rw-r--r-- 1 nobody nobody 1.2M Jan 25 08:51 44-28.ts
-rw-r--r-- 1 nobody nobody  266 Jan 25 08:51 44.m3u8
```

图 3-1

然后通过 VLC 播放 HLS 流媒体，如图 3-2 所示。

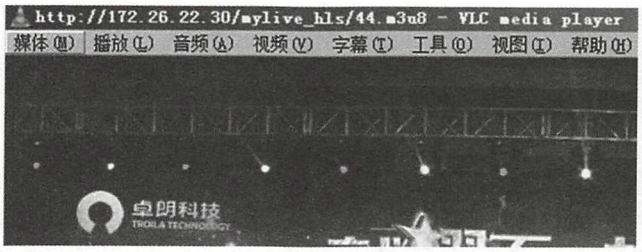


图 3-2

可以看到，通过 HLS 相关配置，我们就可以轻松地得到 m3u8 和 ts 文件了，再通过 location 与 type 中的配置客户端，就可以通过 HTTP 协议访问 m3u8 和 ts 文件播放流媒体了。

3.2　推/拉流与串流码

在直播技术中，经常会涉及推流（Push）和拉流（Pull）。将流媒体推送到流媒体服务器的过程被叫作推流。向服务器获取视频数据的过程就叫作拉流。

```
#将 my.mp4 推送到 RTMP 服务器
ffmpeg -i my.mp4 -vcodec 视频编码库 -acodec 音频编码库 -f -flv rtmp://xx
```

在直播地址中，application 之后的参数就是串流码（Stream Key），它用来区分同一个 application 中不同的直播流。

以下三条互不影响的直播流是三路直播流，虽然第一条与第三条的串流码相同，但是因为 application 不同，所以它们也都是独立且不影响的直播流。

```
1. rtmp://127.0.0.1:1935/live/123
2. rtmp://127.0.0.1:1935/live/234
3. rtmp://127.0.0.1:1935/lv/123
```

3.3　Control 控制器

Control 控制器是 http 模块，它可以通过 HTTP 协议从外部控制 rtmp 模块。通过 Control 控制器，我们可以使用 record、drop 和 redirect 这 3 个命令来实现我们的业务场景。

我们要做的仅仅是将以下 location 添加到 http server 标签中，如图 3-3 所示。这里在 server linten 80 标签下增加了一个 location/control 并设置为 rtmp_control all。这样我们就开启了控制器模块在 http 下的通道。

```
location /control{
            rtmp_control all;
        }
```

```
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    server {
        listen       80;
        server_name  localhost;
        location / {
            root   html;
            index  index.html index.htm;
        }
          location /mylive_hls{
                types {
                        #m3u8 type设置
                        application/vnd.apple.mpegurl m3u8;
                        #ts分片文件设置
                        video/mp2t ts;
                        }
                #指向访问m3u8文件目录
                alias /usr/local/m3u8File;
                add_header Cache-Control no-cache;#禁止缓存
                }
         location /control{
                        rtmp_control all;
                }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
```

图 3-3

3.3.1 record 命令

通过 record 命令，可以实现录制与停止直播流功能。

1. 示例

下面先来看一个最简单的实现录制直播流的配置示例。

```
application mylive{
          live on;
          hls on;
          hls_path /usr/local/m3u8File;
          hls_fragment 2s;
          hls_playlist_length 16s;
          recorder myRecord {
              record all manual;
              record_suffix _.flv;
              record_path /tmp/rec;
          }
      }
```

首先手动创建一个目录 mkdir /tmp/rec，再执行 ffmpeg 推流命令，然后请求开启录制的 URL。

```
ffmpeg -i /tmp/nh.mp4 -c:v libx264 -c:a aac -f flv rtmp://172.26.22.30:1935/mylive/77
#ffmpeg 推流
http://172.26.22.30/control/record/start?app=mylive&name=77&rec=myRecord
#开启录制 URL
```

执行程序后，页面返回/tmp/rec/77_.flv 文件。但是，我们在 tmp/rec 目录下并没发现有录制的文件。这通常是因为授予文件夹的访问权限不够高，无法在 rec 目录中进行创建及写入操作。下面通过 chmod -R 777 /tmp 对目录进行授权：-R 是递归子目录，777 是最高权限，因为 tmp 目录在后面会被放很多东西，所以对其授予全部权限，然后再执行录制命令，成功录制文件，如图 3-4 所示。

```
[root@NRM rec]# ll -h
total 69M
-rw-r--r-- 1 nobody nobody  64M Jan 25 14:19 66.flv
-rw-r--r-- 1 nobody nobody 4.7M Jan 25 14:21 77.flv
```

图 3-4

如何停止录制呢？很简单，将 URL 中的 start 替换为 stop 即可。

2．常用配置

（1）rtmp_stat。

rtmp_stat 是流数据统计模块，在 http 模块中配置它，可以通过 URL 实时监控流媒体的各种状态。

示例如下：

```
#http://ip:post/liveStat
 location /liveStat {
      rtmp_stat all;
      rtmp_stat_stylesheet stat.xsl;
    }
    location /stat.xsl {
      root /download/NRM/;
    }
```

（2）record。

下面是 record 配置中最基础的功能，用来设定录制媒体选项命令，如表 3-1 所示。

表 3-1

命　令	功　能
[off]all|audio|video| keyframes|manual]off	关闭录制音频与视频
all	录制音频与视频
audio	仅录制音频
video	仅录制视频
keyframes	仅记录关键帧
manual	手动开启和关闭录制（默认自动录制）

示例如下：

```
record all manual #手动控制，记录音/视频
record audio #自动控制，记录音频
```

（3）record_path。

record_path 用于设置录制文件的输出路径。

示例如下：

```
record_path /xx/x/
```

（4）record_suffix。

record_suffix 用于设置录制文件输出的文件名。如下代码所示，NRM 会将“{1}”的位置自动替换为 Stream Key，因此，我们可以设置个性的文件名格式规范（支持 strftime 函数）。

示例：

```
record_suffix {1}.flv
```

（5）record_unique。

record_unique 用于将当前时间戳添加到已被记录的文件中，避免在每次产生新记录时，重写文件。record_unique 默认是关闭的，其输出格式为{StreamKey}-{long time}，例如77-1517192930.flv。

示例如下：

```
record_unique on
```

（6）record_append。

record_append 用于将新数据追加到旧文件中，或者当已录制的文件丢失时创建它。旧数据和文件中的新数据之间没有时间差。record_append 默认是关闭的。因此，append 与 unique、suffix 的配置是有所冲突的，因为 unique 与使用了 strftime 函数的 suffix 总会生成一个不同的文件名，因此，append 的追加也就没有意义了。

示例：

```
record_append on
```

（7）record_max_size。

record_max_size 是指最大录制文件的大小，当录制文件超过设置的数值时，文件将会被清空，然后写入后续数据。

示例如下：

```
record_max_size 5000KB
```

3.3.2　drop 命令

在配置了 control 选项后，可以通过 drop 命令有选择地踢出推流用户或拉流用户。

```
#踢出推流用户
http://172.26.22.30/control/drop/publisher?app=mylive&name=777
#踢出全部拉流用户
http://172.26.22.30/control/drop/subscriber?app=mylive&name=777
#根据 IP 踢出拉流用户
http://172.26.22.30/control/drop/client?app=mylive&name=777
&addr=172.26.22.4
#根据序号踢出该拉流用户
http://172.26.22.30/control/drop/client?app=mylive&name=777
&clientid=1
```

3.3.3　redirect 命令

可以通过 redirect 命令有选择地重定向推流用户或拉流用户。

```
#重定向推流地址到 newname
http://172.26.22.30/control/redirect/publisher?app=mylive&name=77&newname=1
#重定向全部拉流用户到新流
http://172.26.22.30/control/redirect/subscriber?app=mylive&name=77&newname=1
```

当然，同 drop 命令一样，也可以通过 clientId 与 addr 来指向地址或用户 ID 进行操作。

3.4　数据统计模块

数据统计模块是 http 模块。因此，统计命令应该位于 http 模块中。

下面在 http server 模块下增加两个 location。

第一个 location 将流媒体的状态全部记录到 stat.xsl 中。第二个 location 将 stat.xsl 访问目录指定到/download/NRM 中。stat.xsl 被包含在 NRM 的安装目录中，直接指向那里就可以。配置完成后，访问 http://172.26.22.30/liveStat，如图 3-5 所示。可以看到页面中清晰地显示了不同直播下的观众、状态、上/下行速率、时间、地址、系统等信息。

```
        location /liveStat {
          rtmp_stat all;
          rtmp_stat_stylesheet stat.xsl;
        }
        location /stat.xsl {
          root /download/NRM/;
        }
```

RTMP	#clients	Video				Audio				In bytes	Out bytes	In bits/s	Out bits/s	State	Time
Accepted: 2		codec	bits/s	size	fps	codec	bits/s	freq	chan	7.58 MB	2.32 MB	3.12 Mb/s	0 Kb/s		19m 32s
mylive															
live streams	2														
77	2	H264 High 4.0	3.04 Mb/s	1920x1080	25	AAC LC	100 Kb/s	48000	2	7.57 MB	2.78 MB	3.14 Mb/s	1.94 Mb/s	active	22s

Id	State	Address	Flash version	Page URL	SWF URL	Dropped	Timestamp	A-V	Time
6	playing	172.26.106.38	LNX 9,0,124,2			0	20080	-11	7s
3	publishing	172.26.22.30	FMLE/3.0 (compatible; Lavf58.5.			0	20080	-11	22s

图 3-5

3.5 Exec 相关功能

可以通过 Exec 下提供的模块来与 shell 命令或 ffmpeg 等常用组件命令进行交互。

当发布开始的时候，触发此事件：

```
    #将当前流推到一个低像素 320px×240px 的新流中
    exec_push ffmpeg -i rtmp://172.26.22.30/mylive/$name -s 320x240 -c:v copy -c:a copy
-f flv rtmp://172.26.22.30/myliveLow/$name
    当拉流开始
exec_pullexecexec_optionsexec_staticexec_kill_signalrespawnrespawn_timeoutexec_publis
hexec_playexec_play_doneexec_publish_doneexec_record_done
```

3.6 本章小结

本章介绍了 HLS 在 NRM 中的使用、推/拉流等术语、control 控制器的使用、直播状态的监控，以及一些常用的配置信息。通过以上内容，我们基本可以搭建出一个支持 HLS 协议的直播服务器，并可以根据常用配置进行个性化设置。

第4章

Nginx-rtmp-module 应用

本章会介绍 FFmpeg 的具体应用与安装配置、NRM 常用配置字典，以及直播系统中常用的架构体系。

4.1 FFmpeg

1. FFmpeg 介绍

FFmpeg 是一个完整的、跨平台的解决方案，用于记录、转换和流化音/视频。

FFmpeg 采用 LGPL 或 GPL 许可证，提供了录制、转换及流化音/视频的完整解决方案。其包含了非常先进的音/视频编解码库 libavcodec。

FFmpeg 是在 Linux 平台下开发的，但是它同样也可以在其他操作系统环境中编译运行，包括 Windows、Mac OS X 等系统。这个项目最早是由 Fabrice Bellard 发起的，在 2004 年至 2015 年由 Michael Niedermayer 主要负责维护。许多 FFmpeg 的开发人员都来自 MPlayer 项目组，而且当前 FFmpeg 也是被放在 MPlayer 项目组的服务器上的。其名称来自 MPEG 视频编码标准，“FF”代表“Fast Forward”。

2. FFmpeg 组件

FFmpeg 的组件包含 libavcodec、libavutil、libavformat、libavfilter、libavdevice、libswscale 和 libswresample（这些都可以应用于应用程序），以及 ffmpeg、ffplay 和 ffprobe（可以被终端用户进行编码和播放），如图 4-1 所示。

```
[root@NRM ~]# ffmpeg -version
ffmpeg version 3.4.git Copyright (c) 2000-2018 the FFmpeg developers
built with gcc 4.4.7 (GCC) 20120313 (Red Hat 4.4.7-18)
configuration: --prefix=/usr/local/ffmpeg --enable-libx264 --enable-gpl
libavutil      56.  7.100 / 56.  7.100
libavcodec     58.  9.100 / 58.  9.100
libavformat    58.  5.101 / 58.  5.101
libavdevice    58.  0.101 / 58.  0.101
libavfilter     7. 11.101 /  7. 11.101
libswscale      5.  0.101 /  5.  0.101
libswresample   3.  0.101 /  3.  0.101
libpostproc    55.  0.100 / 55.  0.100
```

图 4-1

- libavutil 是一个包含简化编程功能的库，包括随机数生成器、数学例程、核心多媒体实用程序等。
- libavcodec 是一个包含解码和编码器的音/视频编解码器的库。
- libavformat 是一个包含用于多媒体容器格式的 demuxers 和 muxers 的库。
- libavdevice 是一个包含输入和输出设备的库，用于抓取和呈现许多常见的多媒体输入/输出软件框架，包括 Video4Linux、Video4Linux2、VfW 和 ALSA。
- libavfilter 是一个包含媒体过滤器的库。
- libswscale 是一个执行高度优化的图像缩放和颜色空间/像素格式转换操作的库。
- libswresample 是一个执行高度优化的音频重采样、rematrixing 和示例格式转换操作的库。
- libpostproc 是一个用于后期效果处理的库。

3．所支持的协议

FFmpeg 所支持的协议包括：HTTP、RTP、RTSP、RealMedia RTSP/RDT、TCP、UDP、Gopher、RTMP、RTMPT、RTMPE、RTMPTE、RTMPS、SDP、MMS over TCP。

4．示例

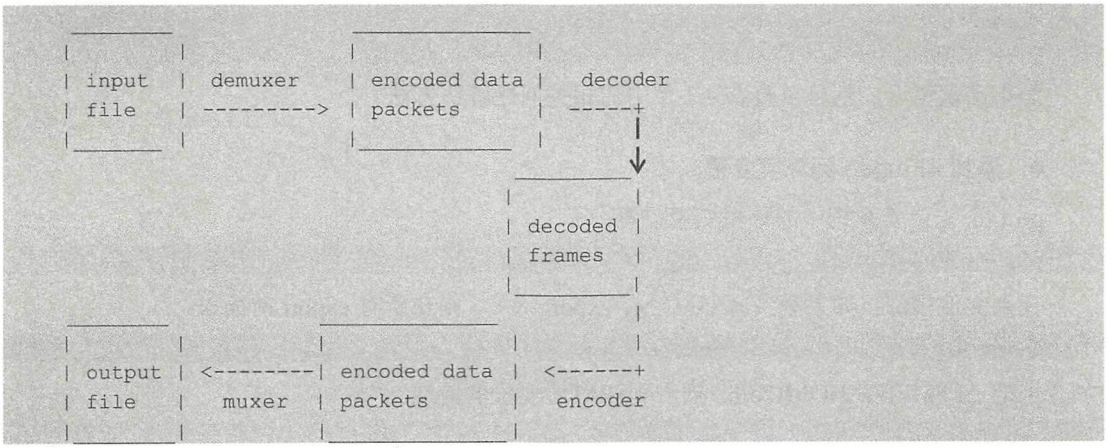
转码一个视频码率为 4Mbps：

```
ffmpeg -i input.avi -b 4Mbps output.avi
```

转码一个视频为 24 帧：

```
ffmpeg -i input.avi -r 24 output.avi
```

转码流程如下：



（1）FFmpeg 调用 libavformat 库（包含 demuxers）来读取输入文件，并获取包含编码数据的数据包。

（2）将编码的信息包传递给解码器。

（3）解码器产生未压缩的帧，可以通过过滤做进一步处理。

（4）过滤后，帧被传递给编码器，编码器将其编码并输出编码的数据包。

（5）最后，这些数据包被传递给 muxer，它将编码的包写入输出文件中。

4.1.1　FFmpeg 的安装

本节介绍 FFmpeg 的安装及组件的选择。

1．下载

先下载 ffmpeg-x.x.x.tar.bz2（本书使用的版本为 ffmpeg-3.4.1.tar.bz2）或通过 wget 命令在 Linux 系统中直接下载。

2．解压

通过 rz 命令将下载文件上传到/download 目录中：

```
tar -xjvf ffmpeg-3.4.1.tar.bz2
mv ffmpeg-3.4.1.tar.bz2 ffmpeg
cd ffmpeg
```

3．配置

进行如下配置：

```
./configure --prefix=/usr/local/ffmpeg
make & make install
```

如果出现错误，则可以查看 4.2 节中的相关问题解决方法。

4．添加 FFmpeg 到环境变量

profile 记录着系统中的环境变量设置：

```
vim /etc/profile
```

找到文件尾部，如果看到尾部已经有 export ××，则在所有 export 前插入：

```
export FFMPEG_HOME=/usr/local/ffmpeg
```

再把 $FFMPEG_HOME/bin: 插入到$PATH 变量的前端。

```
export FFMPEG_HOME=/usr/local/ffmpeg
export PATH=$FFMPEG_HOME/bin:$PATH
```

如果文件尾部没有 exprot，则直接插入，内容如图 4-2 所示。

```
export FFMPEG_HOME=/usr/local/ffmpeg
export JAVA_HOME=/usr/local/java/jdk1.8.0_131/
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/l
export PATH=$FFMPEG_HOME/bin:$JAVA_HOME/bin:$PATH
```

图 4-2

如果在执行 source 命令后提示配置错误，则会导致 profile 文件失效，部分 Linux 命令失效。需要输入“export PATH=/usr/bin:/usr/sbin:/bin:/sbin:/usr/X11R6/bin”，恢复部分 Linux 命令，并尽快修复 profile 文件。

```
source /etc/profile #使之生效
```

shell 命令基本都在/usr/bin，/usr/sbin，/bin，/sbin，/usr/X11R6/bin 目录中定义。

5．ffmpeg ./configure 可能出现的问题

当没有办法创建临时文件到/tmp 目录下时，如图 4-3 所示，可以检查是否有/tmp 目录，如果没有，则输入“mkdir /tmp”即可。

```
[root@NRM ffmpeg-3.4.1]# ./configure --prefix=/usr/local/ffmpeg
Unable to create temporary directory in /tmp.

If you think configure made a mistake, make sure you are using the latest
version from Git.  If the latest version fails, report the problem to the
ffmpeg-user@ffmpeg.org mailing list or IRC #ffmpeg on irc.freenode.net.
Include the log file "ffbuild/config.log" produced by configure as this will help
solve the problem.
```

图 4-3

图 4-4 所示的为没有 NASM/YASM 或其版本太旧导致的问题。

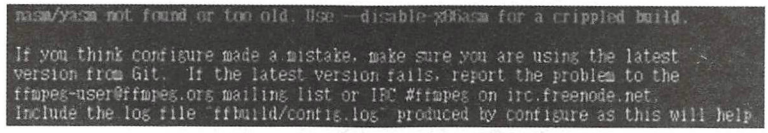


图 4-4

- YASM 是一个完全重写的 NASM 汇编。目前，它支持 x86 和 AMD64 命令集。
- NASM 是一款基于 80×86 和 x86-64 平台的汇编语言编译程序，其设计初衷是为了实现编译器程序跨平台和模块化的特性。

6. 安装 YASM

下载 YASM 的解压文件包，然后配置选项、编译及安装。

```
tar -zxvf yasm-1.3.0.tar.gz
./configure --prefix=/usr/local/yasm
make & make install
```

添加 YASM 到环境变量中：

```
vim /etc/profile
export YASM=/usr/local/yasm/
export PATH 头部追加 $YASM/bin:
source /etc/profile
```

添加后的效果如图 4-5 所示。

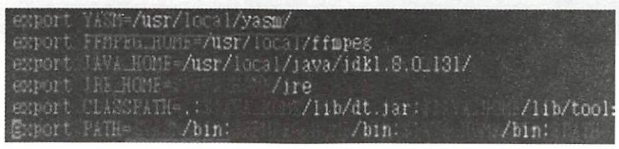


图 4-5

7. 测试

下面进行测试：

```
cd /usr/local/ffmpeg/bin
./ffmpeg -version
```

确认输出的 FFmpeg 版本信息无误，如图 4-6 所示。

8. FFmpeg 验收

下面进行 FFmpeg 验收：

```
ffmpeg -version
```

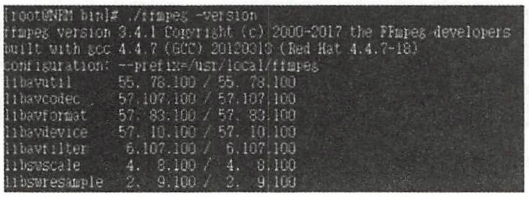


图 4-6

4.1.2 FFmpeg 的配置

FFmpeg 可以选择多种音/视频编码器对媒体进行渲染。下面安装几种常用的视频编码器。

1. 确认组件

因为在 4.1.1 节中是选择默认配置安装 FFmpeg 的，所以我们只有一些列基础库，如图 4-7 所示。

```
ffmpeg -version
```

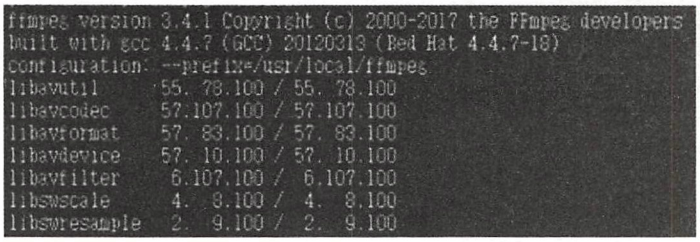


图 4-7

2. libx264

libx264 是当下十分热门的 H264 编码器，有着非常广泛的应用。H264 编码器的优势是低码率、具有流畅连续的高清图像、高容错率、强网络适应性和高压缩比。

H.264 的压缩比是 MPEG-2 的 2 倍以上，是 MPEG-4 的 1.5~2 倍。举一个例子，如果原始文件的大小为 88GB，那么用 MPEG-2 压缩标准压缩后变成 3.5GB，压缩比为 25∶1；用 H.264 压缩标准压缩后变为 879MB，压缩比达到 102∶1。

在下面的命令中增加了–vcodec 和–acodec，这里使用视频编码器 libx264，使用音频编码器 aac。

```
    #ffmpeg x264 的使用
    ffmpeg -i /tmp/nh.mp4 -vcodec libx264 -acodec aac -f flv
rtmp://172.26.22.30:1935/mylive/66
```

3. libx264 安装

重新配置及编译安装 FFmpeg，将新的 libx264 配置到 FFmpeg 中。

```
cd /download/ffmpeg
./configure --prefix=/usr/local/ffmpeg --enable-libx264 --enable-gpl
#启用 libx264，libx264 需要 gpl
#提示 ERROR: libx264 not found，x264 需要我们自己安装，而并不是 FFmpeg 所默认包含的库
```

因为 x264 会依赖 NASM 的汇编加速，因此，这里先安装 NASM。如果不安装 NASM，则会报错：Minimum version is nasm-2.13。

（1）安装 NASM。

安装 NASM 后的结果如图 4-8 所示。

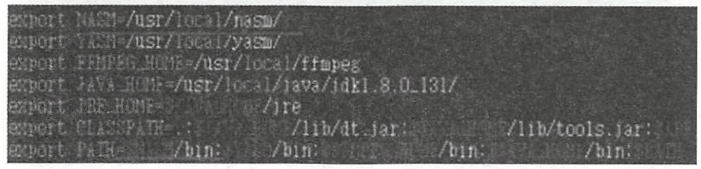


图 4-8

```
下载的 NASM 版本不得低于 2.13 版本
http://www.nasm.us/pub/nasm/releasebuilds/
mkdir /download/nasm
rz nasm-2.13.03rc1.tar.bz2
tar -jxvf nasm-2.13.03rc1.tar.bz2 #解压文件包
mv nasm-2.13.03rc1 nasm
cd nasm
./configure --prefix=/usr/local/nasm #配置安装目录
make & make install #编译与安装

#修改环境变量
vim /etc/profile
添加 export NASM=/usr/local/nasm/
export PATH 头部追加 $NASM/bin:
source /etc/profile #生效设置
```

安装 x264：

```
下载最新版 x264
http://www.videolan.org/developers/x264.html
mkdir /download/x264
rz last_x264.tar.bz2
tar -jxvf last_x264.tar.bz2 #解压文件包
mv x264-snapshot-20180123-2245 x264
cd x264
./configure --prefix=/usr/local/x264 --enable-shared #配置动态库
```

输入“./configure”后会列出配置清单让我们确认，如图 4-9 所示。

```
make & make install #编译与安装
cd /download/ffmpeg
./configure --prefix=/usr/local/ffmpeg --enable-libx264 --enable-gpl
```

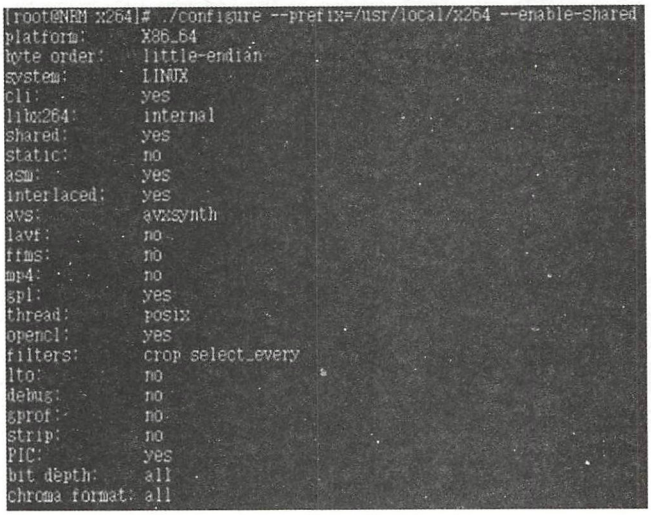


图 4-9

此时依然报错找不到 libx264。

使用“tail /download/ffmpeg/ffbuild/config.log ”命令查看原因，如图 4-10 所示。

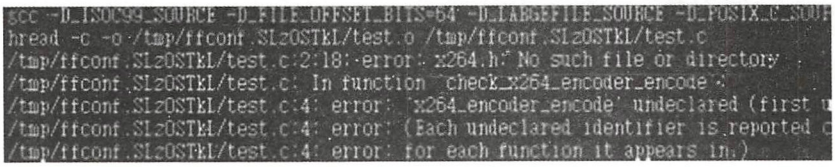


图 4-10

看一下安装 lib x264 时的输出信息，如图 4-11 所示。

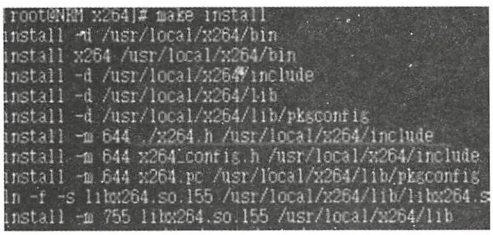


图 4-11

可以看到 x264 文件的目录，因为我们在安装的时候手动选择了安装路径——prefix=/usr/local/x264，因此，x264 目录下的 lib 和 include 目录中的文件无法被 FFmpeg 自动发现。

```
配置 ffmpeg pkgconfig 默认路径的环境变量
让 FFmpeg 可以发现我们自定义的 x264 文件
vim /etc/profile
添加 export PKG_CONFIG_PATH=/usr/local/x264/lib/pkgconfig
#这里就不需要再配置到 PATH 了
./configure --prefix=/usr/local/ffmpeg --enable-libx264 --enable-gpl
```

成功配置并执行后会弹出提示如图 4-12 所示的内容证明安装成功。

```
Enabled indevs:
fbdev                   lavfi

Enabled outdevs:
fbdev                   oss

License: GPL version 2 or later
Creating configuration files ...
libavutil/avconfig.h is unchanged
libavcodec/bsf_list.c is unchanged
libavformat/protocol_list.c is unchanged
```

图 4-12

之后在执行 make&make install 语句时，发现系统依然有报错，如图 4-13 所示。

```
libavcodec/libx264.c: In function 'X264_frame':
libavcodec/libx264.c:282: error: 'x264_bit_depth' undeclared (fir
libavcodec/libx264.c:282: error: (Each undeclared identifier is r
libavcodec/libx264.c:282: error: for each function it appears in.
libavcodec/libx264.c:365: warning: 'coded_frame' is deprecated (d
libavcodec/libx264.c:375: warning: 'coded_frame' is deprecated (d
libavcodec/libx264.c: In function 'X264_init':
libavcodec/libx264.c:533: warning: 'chromaoffset' is deprecated (
libavcodec/libx264.c:534: warning: 'chromaoffset' is deprecated (
libavcodec/libx264.c:547: warning: 'scenechange_threshold' is dep
libavcodec/libx264.c:548: warning: 'scenechange_threshold' is dep
libavcodec/libx264.c:594: warning: 'noise_reduction' is deprecate
libavcodec/libx264.c:595: warning: 'noise_reduction' is deprecate
libavcodec/libx264.c:604: warning: 'b_frame_strategy' is deprecat
libavcodec/libx264.c:605: warning: 'b_frame_strategy' is deprecat
libavcodec/libx264.c:612: warning: 'coder_type' is deprecated (de
libavcodec/libx264.c:613: warning: 'coder_type' is deprecated (de
libavcodec/libx264.c:709: warning: 'me_method' is deprecated (dep
libavcodec/libx264.c:711: warning: 'me_method' is deprecated (dep
libavcodec/libx264.c:713: warning: 'me_method' is deprecated (dep
libavcodec/libx264.c:715: warning: 'me_method' is deprecated (dep
libavcodec/libx264.c:717: warning: 'me_method' is deprecated (dep
libavcodec/libx264.c: In function 'X264_init_static':
libavcodec/libx264.c:892: error: 'x264_bit_depth' undeclared (fir
```

图 4-13

先不要着急，仔细看一下报错信息，原来是 libx264.c 文件中使用的 x264 版本与当前安装

的 x264 版本无法兼容。我们明明是在官网中下载的最新版本的 FFmpeg，怎么会有这样的问题？

因为 x264 与 FFmpeg 是两个不同的开源项目，如果其官网发布打包不及时，则很可能会出现最新版本的 FFmpeg 无法兼容最新的 x264 的情况。

首先进入 x264 官网查看它的 Git 信息以及与 FFmpeg 匹配问题，可以发现 FFmpeg 在 Git 中迭代了这个与 x264 不匹配的问题版本，但是并没有放到官网中。

（2）修复 FFmpeg 与 x264 不匹配的问题。

由此可以确定问题出在 FFmpeg 版本上，那么直接从 Git 官网中下载最新版并且重新安装即可。

```
#删除已安装的 FFmpeg
rm -rf /usr/local/ffmpeg
#删除错误版本的 FFmpeg 安装包与解压缩后目录
rm -rf /download/ffmpeg*
#创建新目录
mkdir /download/ffmpeg
#上传下载好的 git 最新版 FFmpeg
rz FFmpeg-master.zip
#安装 zip 解压命令
yum install -y unzip
unzip FFmpeg-master.zip
```

这里配置应该是成功的，因为在前文中已经配置过 PKG_CONFIG_PATH 环境变量。如果你的代码还是报错，则请仔细看前面的内容。

```
#执行配置命令
./configure --prefix=/usr/local/ffmpeg --enable-libx264 --enable-gpl
make & make install
#安装完毕尝试一下
ffmpeg -i /tmp/nh.mp4 -vcodec libx264 -acodec aac -f flv rtmp://xx
#错误提示
#ffmpeg: error while loading shared libraries:
#libx264.so.155: cannot open shared object file: No such file or directory
#找不到 libx264.so，这个文件在/usr/local/x264/lib 中
#设置动态库
cd /etc/ld.so.conf.d/
mkdir x264.conf
#添加一条信息
/usr/local/x264/lib
#生效
/sbin/ldconfig -v
#配置完毕，ffmpeg libx264 模块已经可以使用了
```

4.1.3　FFmpeg 与直播的应用

前文已经介绍了 NRM 可以通过 exec 来执行各种命令，其中最常见的就是 FFmpeg 命令。可以通过 FFmpeg 命令可以完成几乎我们想要的一切功能。

当触发推流时间时，要将当前流媒体以不同尺寸推到其他的 application 中。

```
    application high{
        live on;
    }
    application middle{
        live on;
    }
    application normal{
        live on;
    }
    application low{
        live on;
    }
    exec_publish  bash -c "ffmpeg -i
    rtmp://172.26.1.92:1935/liveOnline/$name -f flv -r 25 -s 1920x1080 -an
rtmp://172.26.1.92:1935/high/$name -f flv -r 25 -s 1280x720 -an
    rtmp://172.26.1.92:1935/middle/$name -f flv -r 25 -s 640x480 -an
    rtmp://172.26.1.92:1935/normal/$name -f flv -r 25 -s 320*240 -an
    rtmp://172.26.1.92:1935/low/$name";
```

当然，上面的 FFmpeg 命令只是调整了视频尺寸，通常我们还应该配合设置视频帧数来进行制式调整。

4.2　基础配置信息

无论是 rtmp 标签，还是 server 标签，甚至是 application 标签，都可以算是核心配置信息中的成员。因为它们的存在，影响着整个 NRM，必须要配置。

1. rtmp

rtmp 是根级标签，并且是配置中最关键的标签。

```
    rtmp{ ... }
```

2. server

一个 rtmp 中可以包含多个 server 标签，每个 server 标签可以通过端口隔离。

```
    rtmp{
      server{
            listen 1935;
```

```
    }
    server{
        listen 1955;
    }
}
```

3. listen

listen 只能被放在 server 中，指定了所在 server 标签绑定的端口信息。

```
server{
        listen 1955;
    }
```

4. application

application 可以被放在 server 标签中，可以包含多个 application 并通过 applicationName 来隔离。

```
rtmp{
    server{
        listen 1935;
        application A{
        }
        application B{
        }
    }
    server{
        listen 1934;
        application A{}
        application B{}
    }
    #rtmp://ip:1934/A 与 rtmp://ip:1935/A 是 server 隔离
    #rtmp://ip:1934/A 与 rtmp://ip:1934/B 是 application 隔离
}
```

5. ping 和 ping_timeout

ping 和 ping_timeout 可以被放在 rtmp 和 server 中，用于主动检查心跳，将各心跳包发送到客户端。ping_timeout 中设置的值为超时回复时间，如果在超时回复时间内没有得到回复，则关闭客户端。ping 默认为 1 分钟，timeout 默认为 30 秒，当 ping 为 0 时，关闭此功能。

```
ping 15s;
ping_timeout 5s;
```

6. ack_window

ack_window 可以被放在 rtmp 和 server 中，用于设置 rtmp 确认窗口大小，默认为 5000000 字节。

RTMP 消息包一共分成 3 种类型。第一类是命令（通知）消息，第二类是音频消息，第三类是视频消息。而窗口大小则属于第一类消息，即命令消息。窗口大小的本意是让对端了解与本端的通信状况，用以控制媒体传输流量的一种方案。通常，我们从 RTMP 服务器中拉取 RTMP 流到本地时，在协商的过程中，会发送 0x05 和 0x06 消息包，即带宽值通知，通常设为 2.5MB。在实际的拉流过程中，我们通常隔一段时间就得向服务器报告我们已经从服务中收到了多少数据，此种报告就是窗口大小，即 ack size 确认。在实际开发的过程中，通常当接收的数据量接近于 3 倍带宽值（2.5MB×3）时，向服务器报告一下目前已接收了多少数据。

在接收端的 TCP 协议缓存中还有多少剩余空间，发送端必须保证发送的数据不超过这个剩余空间，以免造成缓冲区溢出，这个窗口是接收端用来限制流量的。在传输过程中，窗口大小与接收端的进程取出数据的快慢有关。

```
ack_window 5000000;
```

7．chunk_size

chunk_size 可以被放在 rtmp 和 server 中，用于设置流中的块大小，默认是 4096 字节。这个值越大，CPU 开销就越低，但是这个值不能小于 128 字节。

```
chunk_size 4096;
```

8．max_message

max_message 可以被放在 rtmp 和 server 中，用于设置输入数据报文最大尺寸。所有输入数据都会被分割成报文（然后进一步被分割为块）。报文在处理结束之前会被放在内存中。从理论上讲，如果接收到的报文很大，则可能会影响服务器的稳定性。报文默认值为 1MB，此时可以满足大多数情况。

```
max_message 1M;
```

9．buflen

buflen 可以被放在 rtmp 和 server 中，用于设置缓冲区长度。

```
buflen 5s;
```

10．rtmp_auto_push

rtmp_auto_push 用于设置当多任务进行时，分发任务到多个进程。

```
rtmp_auto_push on;
```

11．rtmp_auto_push_reconnect

rtmp_auto_push_reconnect 用于设置当 rtmp_auto_push 开启并因超时被销毁时，进行重连。

```
rtmp_auto_push_reconnect 1s;
```

12. meta

meta 可以被放在 rtmp，server 和 application 中，用于将元数据信息发送到客户端，默认为打开。

```
meta copy;
context: rtmp, server, application
```

13. interleave

interleave 可以被放在 rtmp，server 和 application 中，用于交叉模式，此模式下音/视频在同一个 chunk stream 上，默认为关闭。

```
interleave on;
```

14. wait_key

wait_key 可以被放在 rtmp，server 和 application 中，用于使视频流从一个关键帧开始，默认为关闭。

```
wait_key on;
```

15. wait_video

wait_video 可以被放在 rtmp，server 和 application 中，用于禁用音频，直到第一个视频帧发送，默认为关闭。可以与 wait_key 结合，使客户端接收视频关键帧。然而，这通常会增加连接延迟。可以在编码器中调整关键帧间隔以减少延迟。最新版本的 IE 浏览器需要设置这个选项才能正常播放。

```
wait_video on;
```

16. sync

sync 可以被放在 rtmp，server 和 application 中，用于同步音频流和视频流。如果客户端带宽不足以接收到服务器的数据，那么一些帧会被服务器删除。这导致了音频流和视频流不同步。当时间戳差异超过指定为同步参数的值时，则将发送一个绝对帧，默认是 300ms。

```
sync 10ms;
```

17. allow,deny

allow,deny 可以被放在 rtmp，server 和 application 中，用于设置白名单和黑名单。

```
allow publish 127.0.0.1;
#允许 127.0.0.1 推流
```

```
deny publish all;
#阻止所有推流，allow publish 中的配置除外
allow play 192.168.0.0/24;
#允许 192.168.0.0/24 拉流
deny play all;
#阻止所有拉流，allow play 的配置除外
```

18. play

play 可以被放在 rtmp，server 和 application 中，用于播放本地或远程点播文件。

```
application vod {
    play /var/flvs;
}

application vod_http {
    play http://myserver.com/vod;
}

application vod_mirror {
    #当第一个地址无法播放的时候，会访问第二个地址
    play /var/local_mirror http://myserver.com/vod;
}
```

19. max_connections

max_connections 可以被放在 rtmp，server 和 application 中，用于设置最大连接数。

```
max_connections 1000;
```

20. access_log

access_log 可以被放在 rtmp，server 和 application 中，用于通常来说，rtmp 日志是和 nginx/logs/access.log 文件存放在一起的，通过 access_log 可以单独存放 rtmp_log。

```
access_log logs/rtmp_access.log
```

21. log_format

log_format：自定义日志格式。

- connection：连接数。
- remote_addr：客户端地址。
- app_application：名称。
- name：最后一个串流码名称。
- args：最后一个播放的流/推流参数。

- flashver：Flash 版本。
- swfurl：swf 地址。
- tcurl：tc 地址。
- pageurl：客户端页面地址。
- command：推/拉流中命令：none，play，publish，play+publish。
- bytes_sent：发送到客户端的字节数。
- bytes_received：接收到客户端的字节数。
- time_local：连接关闭时间。
- session_time：连接持续时间。
- session_readable_time：格式化日期。
- msec：UNIX 时间戳。

```
$remote_addr [$time_local] $command "$app""$name""$args" -
$bytes_received $bytes_sent "$pageurl""$flashver" ($session_readable_time)
```

4.3 本章小结

本章介绍了 FFmpeg 的安装及使用、常用的直播系统架构，以及常见的 Nginx-rtmp-module 配置信息。学完本章，读者应该基本掌握了安装 Nginx-rtmp-module 和 FFmpeg 及两者的配置，并可以建立一个属于自己的直播系统。在后面的内容中会介绍对于不同客户端中 SDK 的开发及应用。

第5章

Android 端解决方案

2017 年，直播成为互联网行业中最抢眼的领域之一。现在，从传统的 PC 端到移动端，各行业都对直播这块“大蛋糕”很感兴趣。

言归正传，下面从技术的角度介绍在 Android 端如何搭建一个直播客户端。

5.1 移动端视频直播介绍

1. 直播行业分析

随着互联网技术的突飞猛进，短短几年，移动设备从开始的只能打电话、发短信和图片的非智能手机，发展为现在的装满了社交、视频、支付、资讯等形形色色应用程序的智能手机，通信内容也从文字、图片扩展到了音频、视频等。

如今正火的视频直播平台，其实在很早以前就出现过，最早的视频聊天室其实就是这种视频直播平台的前身，只是在那个时候主播是依赖计算机进行视频直播的，观众也需要在计算机上观看。现在，随着科技的发展，大多数人都至少有一部智能手机，而且几乎走到哪里都有 Wi-Fi，这些为移动端视频直播奠定了良好的基础。因此，自 2015 年以来，移动端视频直播已经成为众多行业巨头争夺的重点。

2017 年，各大直播平台纷纷获得融资，其中微吼直播已经完成了 C 轮两亿元融资、云犀直播获得 2000 万元 Pre-A 轮融资，百度、腾讯也纷纷直播领域，而卓朗科技作为国内领先的虚拟化和云计算基础软件的企业，也不会放过这次机会。

2. 直播技术的发展趋势

如今，商业化移动直播技术受到各大企业重视。众多互联网直播平台也是层出不穷，但是绝大部分也只是做到了“尽力而为的互动”直播，而不能达到真正的“深层互动”的直播，同时视频清晰度、视频延迟和流畅性等问题也普遍存在。

因此，未来需要一种更为专业的直播技术服务，以及更为完善的解决方案和硬件设备支持，将这种“浅层互动”提升为“深层互动”，以满足各行业客户的定制化需求。我们写这本书不只是想告诉大家我们对直播技术有完善的解决方案，还想通过这本书，把开发直播平台开发技术毫无保留地分享给大家，让每个人都能轻松搭建一个自己的直播平台。关于 RTMP 协议在前面介绍过了，这里就不再做介绍了。本章主要介绍在 Android 端如何完成推流和拉流，我们的推流是借鉴于 GitHub 上的开源项目 Yasea 框架来完成的，拉流是借鉴于 B 站的开源项目 IJKPlayer 框架来完成的，后面会对这两个框架进行详细介绍。

5.2 Yasea 框架介绍

Yasea 是一个 Android 流媒体客户端，它通过移动设备的摄像机和麦克风，把 YUV 和 PCM 的音/视频数据编码并转码为 H.264/AAC 的格式，然后把 H.264/AAC 格式的数据封装到 FLV 中，并通过 RTMP 协议发送到服务器中进行传输。

1. 特性列表

- 支持 Android 4.1 及以上版本。
- 同时支持 H.264/AAC 格式的硬编码和软编码。
- RTMP 推流，事件状态回调。
- 手机横/竖屏动态切换。
- 前/后摄像头热切换。
- 在推流过程随时录制 MP4 格式文件，支持暂停和恢复功能。
- 具有实时美颜（磨皮）滤镜。

2. 框架分析

可以先在 GitHub 上下载源码看一下（在 GitHub 上搜索项目“Yasea”），下载完成后在 Android Studio 中导入源码。在导入过程中可能会出现 SDK 等编译错误，读者可自行修改一下，这里就不详细介绍解决办法了。下面先看一下源码里 library 目录下的 module 的结构，如图 5-1 所示。

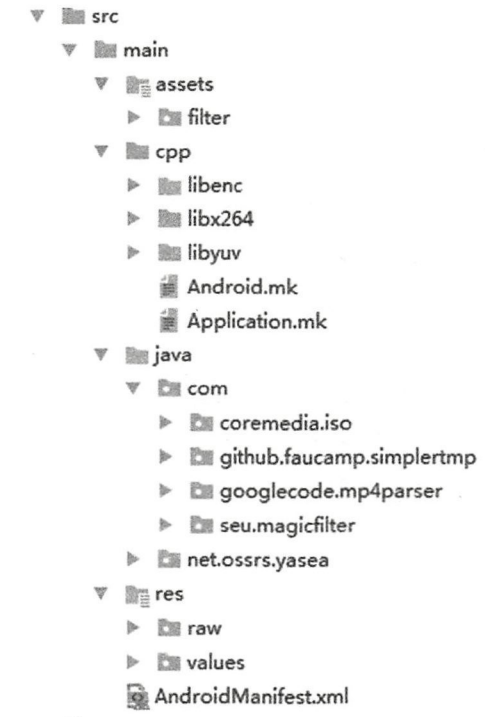


图 5-1

从图 5-1 中可以看出源码大致由 assets、cpp、java、res 这 4 个文件夹组成。其中，assets 文件夹中主要是滤镜需要的图片文件；cpp 文件夹中主要是 JNI 需要的文件；在 java 文件夹中，net.ossrs.yasea 文件夹中是项目封装的核心代码文件，而 com 文件夹中是第三方开源项目 SimpleRtmp、x264、Magicfilter、mp4parser 等代码；在 res 文件夹中，raw 文件夹中是滤镜需要的一些核心文件。

3. 使用介绍

在集成过程中主要用到的是 net.ossrs.yasea 包下 Yasea 框架提供的方法，其他类、方法在这里就不详细介绍了。在具体使用过程中主要用到的是 SrsCameraView 和 SrsPublisher 类中的方法，下面介绍一下这两个类中常用、重要的方法。

```
//设置编码状态的回调方法
public void setEncodeHandler(SrsEncodeHandler handler) {
    mEncoder = new SrsEncoder(handler);
        if (mFlvMuxer != null) {
            mEncoder.setFlvMuxer(mFlvMuxer);
        }
        if (mMp4Muxer != null) {
```

```
                mEncoder.setMp4Muxer(mMp4Muxer);
            }
        }

        //设置录像状态的回调方法
        public void setRecordHandler(SrsRecordHandler handler) {
            mMp4Muxer = new SrsMp4Muxer(handler);
            if (mEncoder != null) {
                mEncoder.setMp4Muxer(mMp4Muxer);
            }
        }

        //设置 RTMP 推流时状态回调
        public void setRtmpHandler(RtmpHandler handler) {
            mFlvMuxer = new SrsFlvMuxer(handler);
            if (mEncoder != null) {
                mEncoder.setFlvMuxer(mFlvMuxer);
            }
        }

        //设置预览分辨率
        public void setPreviewResolution(int width, int height) {
            int resolution[] = mCameraView.setPreviewResolution(width, height);
            mEncoder.setPreviewResolution(resolution[0], resolution[1]);
        }

        //设置推流分辨率
        public void setOutputResolution(int width, int height) {
            if (width <= height) {
                mEncoder.setPortraitResolution(width, height);
            } else {
                mEncoder.setLandscapeResolution(width, height);
            }
        }

        //设置传输率
        public void setVideoHDMode() {
            mEncoder.setVideoHDMode();
        }
        public void setVideoSmoothMode() {
            mEncoder.setVideoSmoothMode();
        }

        //开启美颜功能（其他滤镜效果在 MagicFilterType 中查看，目前提供多种选择方案，如
NONE,FAIRYTALE,SUNRISE,SUNSET,WHITECAT 等）
        public boolean switchCameraFilter(MagicFilterType type) {
            return mCameraView.setFilter(type);
```

```
    }

    //打开摄像头，开始预览（未推流）
    public void startCamera() {
        mCameraView.startCamera();
    }

    //选择软编码
    public void switchToSoftEncoder() {
        mEncoder.switchToSoftEncoder();
    }

    //选择硬编码
    public void switchToHardEncoder() {
        mEncoder.switchToHardEncoder();
    }

    //开始推流 rtmpUrl
    public void startPublish(String rtmpUrl) {
        if (mFlvMuxer != null) {
            mFlvMuxer.start(rtmpUrl);
            mFlvMuxer.setVideoResolution(mEncoder.getOutputWidth(),
mEncoder.getOutputHeight());
            startEncode();
        }
    }

    //stopPublish 停止推流
    public void stopPublish() {
        if (mFlvMuxer != null) {
            stopEncode();
            mFlvMuxer.stop();
        }
    }
```

5.3　IJKPlayer 框架介绍

IJKPlayer 是 B 站的工程师基于 FFmpeg 及 MediaCoder 开发的开源播放器框架，其内部实现了软解码及硬解码的功能，基本流程如图 5-2 所示。

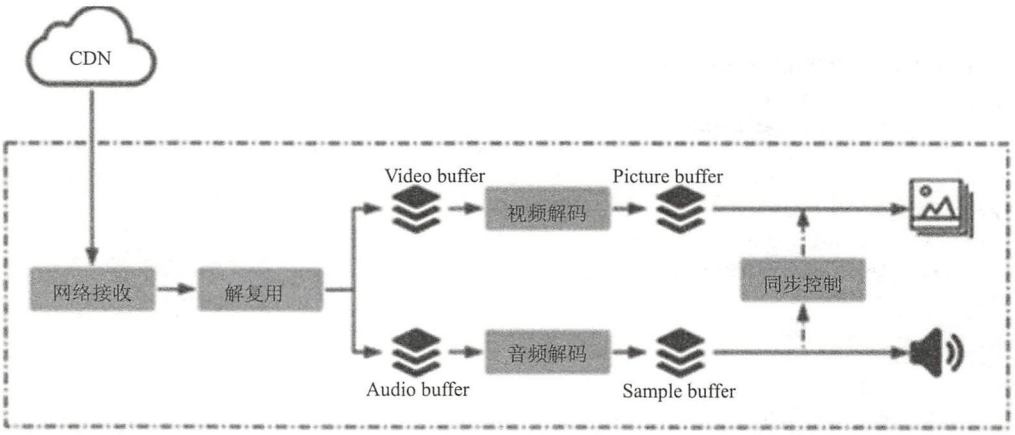


图 5-2

在众多的播放器中，我们选取了比较出众的 IJKPlayer 进行源码剖析。它是一个基于 FFPlay 的轻量级 Android/iOS 端视频播放器框架，具有跨平台的功能，其 API 易于集成、编译配置可裁剪、方便控制安装包大小。

1. 结构说明

可以在 GitHub 上下载或查看 IJKPlayer 项目，其主要目录结构如表 5-1 所示。

表 5-1

目录名称	说　明
tool	初始化项目工程脚本
config	编译 FFmpeg 使用的配置文件
extra	存放编译
ijkplayer	所需的依赖源文件，如 FFmpeg、OpenSSL 等
ijkmedia	核心代码
ijkplayer	播放器数据下载及解码相关
ijksdl	音/视频数据渲染相关
ios	iOS 平台的上层接口封装及平台相关方法
android	Android 平台的上层接口封装及平台相关方法

下面是在 Android 目录中看到的目录名称和说明，如表 5-2 所示。

表 5-2

目录名称	说　明
ijkplayer-arm64	arm64 编译出来的.so 文件
ijkplayer-armv5	armv5 编译出来的.so 文件
ijkplayer-armv7a	armv7a 编译出来的.so 文件
ijkplayer-x86	x86 编译出来的.so 文件
ijkplayer-x86_64	x86_64 编译出来的.so 文件
ijkplayer-exo	Google 开源的一个新的播放器 ExoPlayer，在 Demo 中和在 IJKPlayer 中对比用的。通过安装 IJKPlayer 会发现，在 setting 中可以选择不同播放器来渲染多媒体显示，该模块下面就是一个 Media Player。
ijkplayer-java	IJKPlayer 的一些操作封装及定义。这里面是通用的 API 接口，其中最主要的是 Media Player，它是用来渲染显示多媒体的
ijkplayer-example	IJKPlayer 官方提供的示例

2. 特性列表

如表 5-3 所示的是框架的一些特性和说明。

表 5-3

特性名称	说　明
Android 平台	支持 API 9~23
CPU	ARMv7a，ARM64v8a，x86（ARMv5 未在真实设备上测试）
API	API 提供丰富的方法，易于集成
视频输出	NativeWindow，OpenGL ES 2.0
音频输出	AudioTrack，OpenSL ES
编/解码器	MediaCoder（API 16+，Android 4.1+），支持硬件加速解码，更加省电
其他	可替代 Android.MediaPlayer 和 ExoPlayer

3. 前期准备

在开发直播系统前，先介绍一下如何编译 IJKPlayer 和 FFmpeg。

注意：尽量不要在 Windows 下编译，因为会有各种想不到的 Bug 出现，这里以在 Ubuntu 下编译源码为例。首先必须安装 Homebrew、Git、Yasm 这 3 个软件，然后要注意 Android NDK 必须安装，最好是 r10 之前的版本，不然编译可能会出错。

```
# install homebrew, git, yasm
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew install git
brew install yasm
# add these lines to your ~/.bash_profile or ~/.profile
# export ANDROID_SDK=<your sdk path>
# export ANDROID_NDK=<your ndk path>
# on Cygwin (unmaintained)
# install git, make, yasm
```

全部安装完成后，在命令行中继续一步步执行如下代码。

```
git clone https://github.com/Bilibili/ijkplayer.git ijkplayer-android
cd ijkplayer-android
git checkout -B latest k0.8.4
./init-android.sh
cd android/contrib
./compile-ffmpeg.sh clean
./compile-ffmpeg.sh all
cd ..
./compile-ijk.sh all
```

如果在执行过程中没有出现任何问题，则执行成功后打开android文件夹下的ijkplayer目录，在对应的CPU项目中可以找到相应的.so文件。

4. 简单用法

在官方的示例项目中用到了很多功能，可是其中的大多数功能我们都不需要，所以需要去掉一些没用的代码。exo是Google开发的新的播放器，这里不需要，直接去掉即可。我们需要的只有 tv.danmaku.ijk.media.example.widget.media 包下的部分类。ijkplayer-arm64、ijkplayer-armv5、ijkplayer-armv7a、ijkplayer-x86、ijkplayer-x86_64 是不同体系架构的动态链接库，在工程结构中作为一个模块。如果项目没有要求兼容多平台，则可以删除其他目录结构，单独保留自己需要的平台目录即可。如果对IJKPlayer没有改动，那么也可以在Gradle里直接集成。

```
# required
allprojects {
 repositories {
   jcenter()
 }
}
dependencies {
 # required, enough for most devices.
 compile 'tv.danmaku.ijk.media:ijkplayer-java:0.8.4'
 compile 'tv.danmaku.ijk.media:ijkplayer-armv7a:0.8.4'
 # Other ABIs: optional
 compile 'tv.danmaku.ijk.media:ijkplayer-armv5:0.8.4'
```

```
    compile 'tv.danmaku.ijk.media:ijkplayer-arm64:0.8.4'
    compile 'tv.danmaku.ijk.media:ijkplayer-x86:0.8.4'
    compile 'tv.danmaku.ijk.media:ijkplayer-x86_64:0.8.4'
    # ExoPlayer as IMediaPlayer: optional, experimental
    compile 'tv.danmaku.ijk.media:ijkplayer-exo:0.8.4'
}
```

5. 播放的实现方式

在项目中的 XML 文件中，只需加上下面这个组件即可。

```
<tv.danmaku.ijk.media.widget.media.IjkVideoView
    android:id="@+id/ijkPlayer"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

然后，在 Activity 代码中注册这个组件，并设置其需要的方法。

```
IjkVideoView videoView = (IjkVideoView)
findViewById(R.id.video_view);
  videoView.setAspectRatio(IRenderView.AR_ASPECT_FIT_PARENT);
  String url = "rtmp://192.168.1.92:8080/liveOnline/6666";
  videoView.setVideoURI(Uri.parse(url));
  videoView.start();
```

6. 常用的方法

```
/**
* 参数 aspectRatio(缩放参数)参见 IRenderView 的常量:
IRenderView.AR_ASPECT_FIT_PARENT,
IRenderView.AR_ASPECT_FILL_PARENT,
IRenderView.AR_ASPECT_WRAP_CONTENT,
IRenderView.AR_MATCH_PARENT,
IRenderView.AR_16_9_FIT_PARENT,
IRenderView.AR_4_3_FIT_PARENT
*/
public void setAspectRatio(int aspectRatio);

//改变视频缩放状态
Public 的 int toggleAspectRatio();

//设置视频路径
public void setVideoPath(String path);

//设置视频 URI（可以是网络视频地址）
public void setVideoURI(Uri uri);

//停止视频播放，并释放资源
public void stopPlayback();
```

```
    /**
    * 设置媒体控制器
    * 参数 controller:媒体控制器，注意是
    com.hx.ijkplayer_demo.widget.media.IMediaController。
    */
    public void setMediaController(IMediaController controller);

    //改变媒体控制器的显示和隐藏
    private void toggleMediaControlsVisiblity();

    //注册一个回调函数，在视频预处理完成后调用。在视频预处理完成后被调用。此时已经获取到视频的宽度、高
度、宽高比信息，此时可调用 seekTo 让视频从指定位置开始播放
    public void setOnPreparedListener(OnPreparedListener l);

    //播放完成回调
    public void
    setOnCompletionListener(IMediaPlayer.OnCompletionListener l);

    //播放错误回调
    public void setOnErrorListener(IMediaPlayer.OnErrorListener l);

    //事件发生回调
    public void setOnInfoListener(IMediaPlayer.OnInfoListener l);

    //获取总长度
    public int getDuration();

    //获取当前播放位置
    public long getCurrentPosition();

    //设置播放位置，单位为毫秒
    public void seekTo(long msec);

    //是否正在播放
    public boolean isPlaying();

    //获取缓冲百分比
    public int getBufferPercentage();
```

5.4 Android 端开发实战

通过前面的介绍，相信读者对推流、拉流使用的两个主要框架有了大概的了解。下面介绍一个关于直播系统的实战项目。

5.4.1　主要功能

先来介绍一下此项目要实现的主要功能。常见的直播软件都有两个客户端：直播端和观众端。其中涉及两个功能界面：主播的直播功能界面和观众的播放功能界面。

本项目的直播功能主要包括：

- 美颜设置。
- 摄像头切换。
- 开始/结束推流。
- 软/硬编码转换。

播放功能主要包括：

- 开始/结束拉流。
- 点赞动画。
- 开启/关闭弹幕。

直播功能界面如图 5-3（主播的直播功能界面）和图 5-4（观众的播放功能界面）所示。

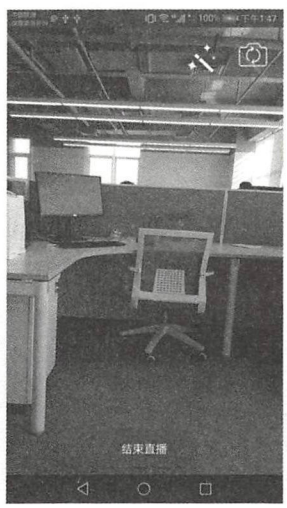


图 5-3

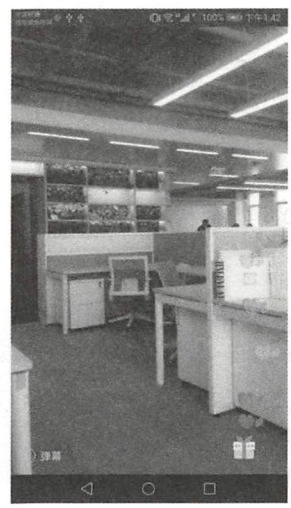


图 5-4

5.4.2　框架导入

为了简捷、方便，在此项目中，除 Yasea 框架外，其他第三方依赖库我们都是在 Gradle 中以引入的方式来实现的，项目结构如图 5-5 所示。

图 5-5

其中主要包括：

- MainActivity：主界面。
- PullActivity：拉流功能代码。
- PushActivity：推流功能代码。

1. 依赖引用说明

```
//Yesea 推流框架引用
implementation project(':YeseaLibrary')

//权限管理
implementation "io.reactivex.rxjava2:rxjava:2.1.9"
implementation 'com.tbruyelle.rxpermissions2:rxpermissions:0.9.5@aar'

//ijkplayer 核心代码
implementation 'tv.danmaku.ijk.media:ijkplayer-java:0.8.4'
//ijkplayer 不同 CPU 下引用
implementation 'tv.danmaku.ijk.media:ijkplayer-armv7a:0.8.4'
implementation 'tv.danmaku.ijk.media:ijkplayer-armv5:0.8.4'
implementation 'tv.danmaku.ijk.media:ijkplayer-arm64:0.8.4'
implementation 'tv.danmaku.ijk.media:ijkplayer-x86:0.8.4'
implementation 'tv.danmaku.ijk.media:ijkplayer-x86_64:0.8.4'
```

```
//弹幕组件核心代码
compile 'com.github.ctiao:DanmakuFlameMaster:0.9.21'
//弹幕不同 CPU 下引用
compile 'com.github.ctiao:ndkbitmap-armv7a:0.9.21'
compile 'com.github.ctiao:ndkbitmap-armv5:0.9.21'
compile 'com.github.ctiao:ndkbitmap-x86:0.9.21'
```

2. 所需权限

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.FLASHLIGHT"/>

<uses-feature android:name="android.hardware.camera.autofocus"/>
<uses-feature android:glEsVersion="0x00020000" android:required="true"/>
```

5.4.3　滤镜

首先介绍一下滤镜功能。我们平时看到直播里的主播都非常漂亮，皮肤状态也非常好，其实这大部分都是滤镜的功劳。直播软件和我们平时用的图片处理工具一样，它也可以有磨皮、美白、增/减亮度、锐化等功能。要在直播中实现滤镜功能，需要用 Yasea 框架中依赖的 MagicCamera 库。MagicCamera 是一个支持实时滤镜的照相机和视频录像的组件，包含美颜等 40 余种实时滤镜，以及具有拍照、录像、修改图片等功能，在 GitHub 中可下载。下面简单介绍一下 MagicCamera 的使用方法。

要实现照相机实时预览功能，只需要在程序的布局文件中集成 MagicCameraView 组件，代码如下：

```
<com.seu.magicfilter.widget.MagicCameraView
    android:id="@+id/glsurfaceview_camera"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

在 Activity 代码中注册 MagicCameraView 组件，通过调用 MagicEngine 对象中的方法来对照相机、滤镜进行操作。

```
publicclassCameraActivityextendsActivity{
    private MagicEngine magicEngine;
    @Override
    protectedvoidonCreate(BundlesavedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_camera);
        MagicEngine.Builderbuilder =  = newMagicEngine.Builder();
```

```
            magicEngine = = builder.build((MagicCameraView)findViewById(R.id.
glsurfaceview_camera));
        }
    }
```

（1）设置滤镜方法。

```
    //参数如下：MagicFilterType.EARLYBIRD
    publicvoidsetCamerFilter(intfilterType){
        magicEngine.setFilter(filterType);
    }
```

（2）设置美颜强度。

```
    publicvoidsetCamerBeautyLevel(intlevel){
        magicEngine.setBeautyLevel(level);
    }
```

（3）保存图片方法。

```
    public void setSavepicture(File file, SavePictureTask.OnPictureSaveListener
listener){
        magicEngine.savePicture(file, listener);
    }
```

（4）设置切换前/后摄像头。

```
    public void setSwitchCamera(){
        magicEngine.switchCamera();
    }
```

（5）开启/关闭录制视频。

```
    //开启录制视频；如果没有设置视频地址，则默认保存地址是 SD 卡根目录
    public void setStartRecord(){
        magicEngine.startRecord();
    }
    //关闭录制视频
    public void setStopRecord(){
        magicEngine.stopRecord();
    }
```

MagicCamera 部分滤镜与 Instagram 的样式对照如表 5-4 所示。

表 5-4

MagicCamera	Instagram
MagicAmaroFilter	Instagram 中 Amaro 滤镜
MagicAntiqueFilter	复古滤镜
MagicBlackCatFilter	黑猫滤镜，增强阴影与色调，颜色加深
MagicBrannanFilter	Instagram 中 Brannan 滤镜
MagicBrooklynFilter	Instagram 中 Brooklyn 滤镜

续表

MagicCamera	Instagram
MagicCalmFilter	平静滤镜，偏棕灰色
MagicCoolFilter	冰冷滤镜，偏蓝色
MagicEarlyBirdFilter	Instagram 中 EarlyBird 滤镜
MagicEmeraldFilter	祖母绿滤镜
MagicEvergreenFilter	常青滤镜
MagicFairytaleFilter	童话滤镜
MagicFreudFilter	Instagram 中 Freud 滤镜
MagicHealthyFilter	健康滤镜
MagicHefeFilter	Instagram 中 Hefe 滤镜
MagicHudsonFilter	Instagram 中 Hudson 滤镜
MagicInkwellFilter	Instagram 中 Inkwell 滤镜
MagicKevinFilter	Instagram 中 Kevin 滤镜
MagicLatteFilter	拿铁滤镜
MagicLomoFilter	Instagram 中 Lomo 滤镜
MagicN1977Filter	Instagram 中 N1977 滤镜
MagicNashvilleFilter	Instagram 中 Nashville 滤镜
MagicNostalgiaFilter	怀旧滤镜，偏绿色
MagicPixarFilter	Instagram 中 Pixar 滤镜
MagicRiseFilter	Instagram 中 Rise 滤镜
MagicRomanceFilter	浪漫滤镜，粉红色系
MagicSakuraFilter	樱花滤镜，粉红色系
MagicSierraFilter	Instagram 中 Sierra 滤镜
MagicSkinWhitenFilter	美白滤镜，皮肤增白
MagicSunriseFilter	日出滤镜
MagicSunsetFilter	日落滤镜
MagicSutroFilter	Instagram 中 Sutro 滤镜
MagicSweetsFilter	甜美滤镜
MagicTenderFilter	温和滤镜
MagicToasterFilter	Instagram 中 Toaster 滤镜
MagicValenciaFilter	Instagram 中 Valencia 滤镜
MagicWarmFilter	温暖滤镜
MagicWhiteCatFilter	白猫滤镜
MagicXproIIFilter	Instagram 中 XproII 滤镜

5.4.4 推流

推流功能依赖的是 Yasea 框架，在前面已经介绍过了。下面看一看如何在代码中实现。先在项目中导入依赖库，然后在布局文件中引用照相机组件，代码如下：

```
    <net.ossrs.yasea.SrsCameraView
            android:id="@+id/pushViewCameraID"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_alignParentBottom="true"
            android:layout_alignParentEnd="true"
            android:layout_alignParentLeft="true"
            android:layout_alignParentRight="true"
            android:layout_alignParentStart="true"
            android:layout_alignParentTop="true" />
```

在 Activity 里进行组件注册、功能设置等：

```
    public class PushActivity extends AppCompatActivity implements
RtmpHandler.RtmpListener,
            SrsRecordHandler.SrsRecordListener, SrsEncodeHandler.SrsEncodeListener{
     private SrsPublisher mPublisher;//推流对象
     private  ImageView btuTurnCameraID;

      @Override
        protected void onCreate(@Nullable Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_push_view);

           SrsCameraView srsCameraView = findViewById(R.id.pushViewCameraID);//相机组件
           btuTurnCameraID = findViewById(R.id.btuTurnCameraID);

           mPublisher = new SrsPublisher(srsCameraView);
            //设置编码状态的回调方法
            mPublisher.setEncodeHandler(new SrsEncodeHandler(this));
            //设置 RTMP 推流时状态回调
            mPublisher.setRtmpHandler(new RtmpHandler(this));
            //设置录像状态的回调方法
            mPublisher.setRecordHandler(new SrsRecordHandler(this));
            //设置预览分辨率
            mPublisher.setPreviewResolution(1920, 1080);
            //设置推流分辨率
            mPublisher.setOutputResolution(1080, 1920);
            //设置传输率
            mPublisher.setVideoHDMode();
            //设置直播源
            mPublisher.startPublish(urlStr);
            //开启摄像头
```


```
            mPublisher.startCamera();

             //切换前后摄像头
            btuTurnCameraID.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    mPublisher.switchCameraFace((mPublisher.getCamraId() + 1) %
Camera.getNumberOfCameras());
                }
            });
        }
    }
```

在上面的代码中可以看到，我们分别对编码、RTMP、录像设置了状态回调方法，从而可以在这些方法中获得是否连接成功、是否断开、视频码率、帧率状态等。

```
    //网络弱
        @Override
        public void onNetworkWeak() {}
        //网络重新连接
        @Override
        public void onNetworkResume() {}
        //编码出现 IllegalArgumentException 错误
        @Override
        public void onEncodeIllegalArgumentException(IllegalArgumentException e) {}
        //录像暂停
        @Override
        public void onRecordPause() {}
        //录像重新开始
        @Override
        public void onRecordResume() {}
        //录像开始
        @Override
        public void onRecordStarted(String msg) {
            Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_SHORT).show();
        }
        //录像关闭
        @Override
        public void onRecordFinished(String msg) {
            Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_SHORT).show();
        }
        //录像出现 IllegalArgumentException 错误
        @Override
        public void onRecordIllegalArgumentException(IllegalArgumentException e) {}
        //录像出现 IOException 错误
        @Override
        public void onRecordIOException(IOException e) {}
        //RTMP 连接
```


```
        @Override
        public void onRtmpConnecting(String msg) {
            Toast.makeText(getApplicationContext(), "连接中。。。", Toast.LENGTH_SHORT).show();
        }
        //RTMP 连接成功
        @Override
        public void onRtmpConnected(String msg) {
            Toast.makeText(getApplicationContext(), "连接成功，您的 Show Time",
Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onRtmpVideoStreaming() {}

        @Override
        public void onRtmpAudioStreaming() {}
        //流媒体结束
        @Override
        public void onRtmpStopped() {}

        //流媒体连接
        @Override
        public void onRtmpDisconnected() {
            Toast.makeText(getApplicationContext(), "直播断开连接",
Toast.LENGTH_SHORT).show();
        }

        //流媒体 FPS 改变时
        @Override
        public void onRtmpVideoFpsChanged(double fps) {
            Log.i(TAG, String.format("Output Fps: %f", fps));
        }
        //流媒体视频码率(Bitrate)状态改变时
        @Override
        public void onRtmpVideoBitrateChanged(double bitrate) {
            int rate = (int) bitrate;
            if (rate / 1000 > 0) {
                Log.i(TAG, String.format("Video bitrate: %f kbps", bitrate / 1000));
            } else {
                Log.i(TAG, String.format("Video bitrate: %d bps", rate));
            }
        }
        //音/视频码率状态改变时
        @Override
        public void onRtmpAudioBitrateChanged(double bitrate) {
            int rate = (int) bitrate;
            if (rate / 1000 > 0) {
```

```
            Log.i(TAG, String.format("Audio bitrate: %f kbps", bitrate / 1000));
        } else {
            Log.i(TAG, String.format("Audio bitrate: %d bps", rate));
        }
    }
    //RTMP Socket 错误
    @Override
    public void onRtmpSocketException(SocketException e) {}
    //RTMP IOException 错误
    @Override
    public void onRtmpIOException(IOException e) {}

    @Override
    public void onRtmpIllegalArgumentException(IllegalArgumentException e) {}

    @Override
    public void onRtmpIllegalStateException(IllegalStateException e) {}
```

5.4.5　拉流

此项目中的拉流功能使用的是 B 站开源的 IJKPlayer 框架。在前面介绍过 IJKPlayer 框架编译和它的一些方法。IJKPlayer 是一个基于 FFmpeg 的轻量级 Android 视频播放器，而 FFmpeg 是全球领先的多媒体框架之一，它能够解码、编码、转码、流和播放大部分的视频格式。

1. 环境配置

这里使用在 Gradle 中引入 IJKPlayer 的方式：

```
# required
allprojects {
    repositories {
        jcenter()
    }
}

dependencies {
    # required, enough for most devices.
    compile 'tv.danmaku.ijk.media:ijkplayer-java:0.8.8'
    compile 'tv.danmaku.ijk.media:ijkplayer-armv7a:0.8.8'

    # Other ABIs: optional
    compile 'tv.danmaku.ijk.media:ijkplayer-armv5:0.8.8'
    compile 'tv.danmaku.ijk.media:ijkplayer-arm64:0.8.8'
    compile 'tv.danmaku.ijk.media:ijkplayer-x86:0.8.8'
    compile 'tv.danmaku.ijk.media:ijkplayer-x86_64:0.8.8'
```

```
        # ExoPlayer as IMediaPlayer: optional, experimental
        compile 'tv.danmaku.ijk.media:ijkplayer-exo:0.8.8'
    }
```

2. 播放器使用

依赖的框架集成完之后，根据 IJKPlayer 提供的 IMediaPlayer 类，自定义一个播放器组件：

```
    public class VideoPlayerIJK extends FrameLayout {

        //由 IJKplayer 提供，用于播放视频，需要给其传入一个 surfaceView
        private IMediaPlayer mMediaPlayer = null;

        //视频文件地址
        private String mPath = "";

        private SurfaceView surfaceView;

        private VideoPlayerListener listener;
        private Context mContext;

        public VideoPlayerIJK(@NonNull Context context) {
            super(context);
            initVideoView(context);
        }

        public VideoPlayerIJK(@NonNull Context context, @Nullable AttributeSet attrs) {
            super(context, attrs);
            initVideoView(context);
        }

        public VideoPlayerIJK(@NonNull Context context, @Nullable AttributeSet attrs,
@AttrRes int defStyleAttr) {
            super(context, attrs, defStyleAttr);
            initVideoView(context);
        }

        private void initVideoView(Context context) {
            mContext = context;

            setFocusable(true);
        }

        /**
         * 设置视频地址
         * 根据是否第一次播放视频，做不同的操作
         * @param path 视频地址
         */
```

```
    public void setVideoPath(String path) {
        if (TextUtils.equals("", mPath)) {
            //如果是第一次播放视频，则创建一个新的 surfaceView
            mPath = path;
            createSurfaceView();
        } else {
            //否则就直接导入
            mPath = path;
            load();
        }
    }

    // 新建一个 surfaceview
    private void createSurfaceView() {
        //生成一个新的 surface view
        surfaceView = new SurfaceView(mContext);
        surfaceView.getHolder().addCallback(new LmnSurfaceCallback());
        LayoutParams layoutParams = new LayoutParams(LayoutParams.MATCH_PARENT
                , LayoutParams.MATCH_PARENT, Gravity.CENTER);
        surfaceView.setLayoutParams(layoutParams);
        this.addView(surfaceView);
    }

    //surfaceView 的监听器
    private class LmnSurfaceCallback implements SurfaceHolder.Callback {
        @Override
        public void surfaceCreated(SurfaceHolder holder) {
        }

        @Override
        public void surfaceChanged(SurfaceHolder holder, int format, int width, int
height) {
            //surfaceview 创建成功后，加载视频
            load();
        }

        @Override
        public void surfaceDestroyed(SurfaceHolder holder) {
        }
    }

    //加载视频
    private void load() {
        //每次都要重新创建 IMediaPlayer
        createPlayer();
        try {
            mMediaPlayer.setDataSource(mPath);
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
        //给 mediaPlayer 设置视图
        mMediaPlayer.setDisplay(surfaceView.getHolder());

        mMediaPlayer.prepareAsync();
    }

    //创建一个新的 player
    private void createPlayer() {
        if (mMediaPlayer != null) {
            mMediaPlayer.stop();
            mMediaPlayer.setDisplay(null);
            mMediaPlayer.release();
        }
        IjkMediaPlayer ijkMediaPlayer = new IjkMediaPlayer();
        ijkMediaPlayer.native_setLogLevel(IjkMediaPlayer.IJK_LOG_DEBUG);

        //开启硬解码
        ijkMediaPlayer.setOption(IjkMediaPlayer.OPT_CATEGORY_PLAYER, "mediacodec", 1);

        mMediaPlayer = ijkMediaPlayer;

        if (listener != null) {
            mMediaPlayer.setOnPreparedListener(listener);
            mMediaPlayer.setOnInfoListener(listener);
            mMediaPlayer.setOnSeekCompleteListener(listener);
            mMediaPlayer.setOnBufferingUpdateListener(listener);
            mMediaPlayer.setOnErrorListener(listener);
        }
    }

    public void setListener(VideoPlayerListener listener) {
        this.listener = listener;
        if (mMediaPlayer != null) {
            mMediaPlayer.setOnPreparedListener(listener);
        }
    }

    //开始播放
    public void start() {
        if (mMediaPlayer != null) {
            mMediaPlayer.start();
        }
    }
```

```
//释放播放器
public void release() {
    if (mMediaPlayer != null) {
        mMediaPlayer.reset();
        mMediaPlayer.release();
        mMediaPlayer = null;
    }
}

//暂停播放
public void pause() {
    if (mMediaPlayer != null) {
        mMediaPlayer.pause();
    }
}

//停止播放
public void stop() {
    if (mMediaPlayer != null) {
        mMediaPlayer.stop();
    }
}

//重新播放
public void reset() {
    if (mMediaPlayer != null) {
        mMediaPlayer.reset();
    }
}

//获取时长
public long getDuration() {
    if (mMediaPlayer != null) {
        return mMediaPlayer.getDuration();
    } else {
        return 0;
    }
}

//获取当前播放位置
public long getCurrentPosition() {
    if (mMediaPlayer != null) {
        return mMediaPlayer.getCurrentPosition();
    } else {
        return 0;
    }
}
```

```
    //跳到指定播放位置
    public void seekTo(long l) {
        if (mMediaPlayer != null) {
            mMediaPlayer.seekTo(l);
        }
    }
}
```

自定义完成后，在布局文件中加入播放器控件：

```
<com.troila.live.demo.widget.VideoPlayerIJK
        android:id="@+id/videoViewID"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center"/>
```

Activity 中的注册界面、组件，功能设置如下：

```
public class PullActivity extends AppCompatActivity{
 @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pull_view);

        String urlStr = "RMTP:视频地址";

         //加载 so 文件
        try {
            IjkMediaPlayer.loadLibrariesOnce(null);
            IjkMediaPlayer.native_profileBegin("libijkplayer.so");
        } catch (Exception e) {
            this.finish();
        }

        VideoPlayerIJK ijkPlayer = findViewById(R.id.videoViewID);
        ijkPlayer.setVideoPath(urlStr);

        //播放器设置监听方法
        ijkPlayer.setListener(new VideoPlayerListener() {
            @Override
            public void onBufferingUpdate(IMediaPlayer mp, int percent) {
            }

            @Override
            public void onCompletion(IMediaPlayer mp) {
                mp.seekTo(0);
                mp.start();
            }
```

```
            @Override
            public boolean onError(IMediaPlayer mp, int what, int extra) {
                return false;
            }

            @Override
            public boolean onInfo(IMediaPlayer mp, int what, int extra) {
               if(what == IMediaPlayer.MEDIA_INFO_BUFFERING_START){
                    textNoDataID.setVisibility(View.VISIBLE);
                    ijkPlayer.setVisibility(View.GONE);
                    Toast.makeText(PullActivity.this,"直播结束",
Toast.LENGTH_SHORT).show();
                }else if(what ==  IMediaPlayer.MEDIA_INFO_VIDEO_RENDERING_START
                        || what ==  IMediaPlayer.MEDIA_INFO_BUFFERING_END){
                    textNoDataID.setVisibility(View.GONE);
                    ijkPlayer.setVisibility(View.VISIBLE);
                   Toast.makeText(PullActivity.this,"直播开始",
Toast.LENGTH_SHORT).show();
                }
                return false;
            }

            @Override
            public void onPrepared(IMediaPlayer mp) {
                mp.start();
            }

            @Override
            public void onSeekComplete(IMediaPlayer mp) {

            }

            @Override
            public void onVideoSizeChanged(IMediaPlayer mp, int width, int height, int
sar_num, int sar_den) {
                //获取到视频的宽和高的尺寸
            }
        });
    }

    @Override
     protected void onStop() {
        super.onStop();
        IjkMediaPlayer.native_profileEnd();
     }
```

5.4.6 弹幕

为了方便用户拓展直播系统的功能，下面在示例项目中加入一些简单的功能，比如点赞、弹幕等。下面先介绍一下同样是由 B 站开源的 Danmaku Flame Master 弹幕解析绘制框架。它也是目前 Android 平台中最好的弹幕框架，目前已经被优酷、土豆、斗鱼、AcFun 等 App 使用。

1. 特性

Danmaku Flame Master 框架具有以下特性。

- 支持使用多种方式（View/SurfaceView/TextureView）实现高效绘制。
- 支持 B 站 XML 弹幕格式解析。
- 支持基础弹幕精确还原绘制。
- 支持 Mode 7 特殊弹幕功能。
- 支持多核机型优化，高效预缓存机制。
- 支持多种显示效果选项实时切换。
- 支持实时弹幕显示。
- 支持换行弹幕支持/运动弹幕。
- 支持自定义字体。
- 支持多种弹幕参数设置。
- 支持多种方式的弹幕屏蔽。

2. 集成方法

在 Gradle 中引入的方式：

```
repositories {
    jcenter()
}

dependencies {
    compile 'com.github.ctiao:DanmakuFlameMaster:0.9.21'
    compile 'com.github.ctiao:ndkbitmap-armv7a:0.9.21'

    # Other ABIs: optional
    compile 'com.github.ctiao:ndkbitmap-armv5:0.9.21'
    compile 'com.github.ctiao:ndkbitmap-x86:0.9.21'
}
```

在文件中加入弹幕控件：

```
<master.flame.danmaku.ui.widget.DanmakuView
```

```
        android:id="@+id/danmakuViewID"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

Activity 中的功能代码也简单，注册弹幕控件并添加数据即可：

```
public class PullActivity extends AppCompatActivity{

    private DanmakuView mDanmakuView;//弹幕组件
    private DanmakuContext danmakuContext;//弹幕内容

  @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pull_view);

        mDanmakuView = findViewById(R.id.danmakuViewID);
        //初始化弹幕组件
        initDanmaku();
    }

     //初始化弹幕组件
    private void initDanmaku() {
        //给弹幕视图设置回调，在准备阶段获取弹幕信息并开始
        mDanmakuView.setCallback(new DrawHandler.Callback() {
            @Override
            public void prepared() {
                mDanmakuView.start();
            }

            @Override
            public void updateTimer(DanmakuTimer timer) {

            }

            @Override
            public void danmakuShown(BaseDanmaku danmaku) {

            }

            @Override
            public void drawingFinished() {

            }
        });
        //缓存，提升绘制效率
        mDanmakuView.enableDanmakuDrawingCache(true);
        //DanmakuContext 主要用于弹幕样式的设置
```

```
            danmakuContext = DanmakuContext.create();
            danmakuContext.setDanmakuStyle(IDisplayer.DANMAKU_STYLE_STROKEN,3);//描边
            danmakuContext.setDuplicateMergingEnabled(false);//重复合并
            danmakuContext.setScrollSpeedFactor(1.2f);//弹幕滚动速度
            //让弹幕进入准备状态，传入弹幕解析器和样式设置
            mDanmakuView.prepare(parser,danmakuContext);
            //显示 FPS、时间等调试信息
            mDanmakuView.showFPS(false);
        }

        // 随机生成一些弹幕内容以供测试
        private void generateSomeDanmaku() {

            for (int i = 0 ; i < 100 ; i ++){
                int time = new Random().nextInt(300);
                String content = "" + time + time;
                addDanmaku(content, false);
            }
        }


        /**
         * 向弹幕 View 中添加一条弹幕
         * @param content    弹幕的具体内容
         * @param withBorder 弹幕是否有边框
         */
        private void addDanmaku(String content, boolean withBorder) {
            //弹幕实例 BaseDanmaku，传入参数是弹幕方向
            BaseDanmaku danmaku =
danmakuContext.mDanmakuFactory.createDanmaku(BaseDanmaku.TYPE_SCROLL_RL);
            danmaku.text = content;
            danmaku.padding = 5;
            danmaku.textSize = sp2px(20);
            danmaku.textColor = Color.WHITE;
            int time = new Random().nextInt(900);
            danmaku.setTime(mDanmakuView.getCurrentTime() + time*10);
            //加边框
            if (withBorder) {
                danmaku.borderColor = Color.GREEN;
            }
            mDanmakuView.addDanmaku(danmaku);
        }

        @Override
        protected void onPause() {
            super.onPause();
            if (mDanmakuView != null && mDanmakuView.isPrepared()) {
```

```
            mDanmakuView.pause();
        }
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (mDanmakuView != null && mDanmakuView.isPrepared() &&
mDanmakuView.isPaused()) {
            mDanmakuView.resume();
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        //退出时释放资源
        if (mDanmakuView != null) {
            mDanmakuView.release();
            mDanmakuView = null;
        }
    }
```

效果如图 5-6 所示。

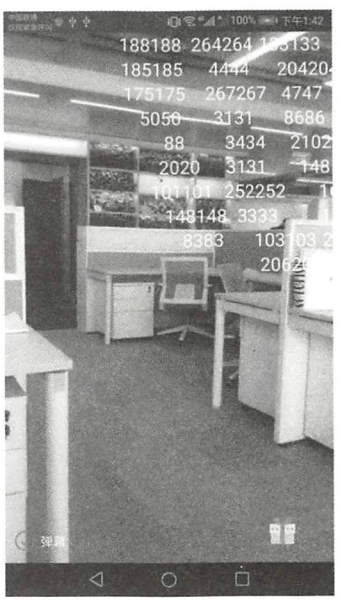


图 5-6

5.5 本章小结

本章介绍了在 Android 端实现直播的过程，同时也对 Yasea、IJKPlayer、Magicfilter、DanmakuFlameMaster 等开源框架做了简单的说明。

这些框架为实现直播系统这些强大的功能奠定了良好的基础，在这里，我们也要感谢这些框架开发者的无私奉献。

看到这里，如果读者还没有能成功搭建一个自己的直播平台，那么也可以看看我们提供的直播 Demo。当然这几个框架的功能远远不止这些，如果你想更深入地了解这些框架，那么可以到 GitHub 上关注一下。

第6章

iOS 端解决方案

前 5 章介绍了如何搭建视频直播服务器，以及在 Android 端如何实现视频直播。本章介绍在 iOS 端如何实现视频直播。

6.1 iOS 端视频直播介绍

在介绍如何在 iOS 端实现视频直播之前，首先要介绍在手机端实现视频直播是通过哪几步实现的。

1. 手机端和服务器端的交互

向服务器端传输数据的过程被称作推流。向服务器端获取视频数据的过程被称为拉流。

2. 手机端视频直播流程

手机端视频直播流程大体分为采集、前期处理、编码、推流和传输、服务器处理、解码和拉流、播放这 7 步。

（1）采集。

采集是整个视频直播流程中的第一个环节，它是从设备中获取原始视频和音频数据，然后将其输出到下一个环节中。视频的采集涉及两个方面的数据采集：音频采集和图像采集，它们分别对应两种完全不同的输入源和数据格式。

（2）前期处理。

现在的直播不仅仅是简单地录制视频，一般还需要对视频做一些美化处理，比如美颜、添加模糊效果及水印等。目前，在 iOS 端，在这方面最著名的开源框架就是 GPUImage。其中内置了 125 种渲染效果，还支持各种自定义脚本等，让我们可以对视频做更多的处理。

（3）编码。

对流媒体传输来说，编码是比较重要的，编码性能、编码速度和编码压缩比会直接影响整个流媒体传输过程中的用户体验和传输成本。编码的重点和难点在于要在分辨率、帧率、GOP 等参数设计上找到最佳平衡点。iOS 8 被推出之后，苹果公司开放了 VideoToolbox.framework，其硬件兼容性比较好，用户可以直接采取硬编码，常用的编码有 H.265 等。

（4）推流和传输。

推流和传输取决于服务器端的性能，发送端和接收端的网络连接、抖动和缓存还是需要客户端来实现的。目前的主要传输协议一般是 RTMP、HLS、FLV 等。

（5）服务器处理。

服务器会对推来的视频流进行流处理，以适配各种不同协议，例如 RTMP、HLS、FLV 等。

（6）解码和拉流。

编码和解码是相对存在的，推流需要编码，同样，拉流需要解码。

（7）播放。

播放器会对流进行播放。

6.2 SDK 的选择和前期准备

前面介绍了实现手机端视频直播一共分为 7 步，其中采集、编码和解码可以直接调用 iOS 端的 SDK，相对比较简单。而前期处理、推流和拉流是相对比较复杂的操作，如果你有底层编码经验，则可以直接针对不同协议进行开发。如果你是初学者或者对底层开发不是很了解，则可以直接选择一些开源库进行开发。我们经过多方对比，下面为读者选择了一些比较成熟的开源库，在此也对提供这些开源库的作者表示感谢。

1. SDK 的选择

（1）GPUImage。

GPUImage 是一款强大的图像处理框架，它是基于 OpenGL ES 的开源框架，其中提供了各种各样的图像处理滤镜，并且支持照相机和摄像机的实时滤镜功能，还能够自定义图像滤镜。

我们可以利用这个框架对视频进行前期处理。

（2）LFLiveKit。

LFLiveKit 是优酷土豆旗下开源的 iOS 推流框架，可以利用这个框架进行推流操作。

（3）IJKPlayer。

IJKPlayer 是 B 站提供的开源拉流框架，它同时支持 iOS 和 Android 两个客户端，可以利用这个框架进行拉流操作。

2. 前期准备

在正式开发项目之前，还需要做一些准备工作。

（1）安装 Yasm。

下载最新版本的 Yasm 安装文件（yasm-1.3.0.tar）。

打开安装文件压缩包，在终端依次执行如下命令：

```
cd yasm-1.3.0
 ./configure
 make
 sudo make install
 输入密码并执行
```

执行完命令会出现如图 6-1 所示的效果。

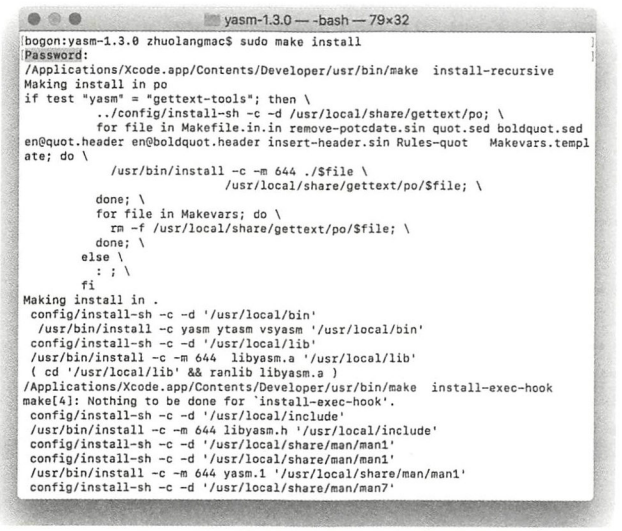


图 6-1

（2）下载 gas-preprocessor 文件。

下载 gas-preprocessor.pl 文件。

（3）安装 Brew。

在终端输入如下命令：

```
curl -LsSf http://github.com/mxcl/homebrew/tarball/master | sudo tar xvz -C/usr/local
--strip 1
```

（4）编译 FFmpeg。

FFmpeg 是一套可以用来记录、转换数字音/视频，并能将其转换为流的开源计算机程序。其采用 LGPL 或 GPL 许可证，并提供了录制、转换及流化音/视频的完整解决方案。在完成前面 3 项准备工作后就可以编译 FFmpeg 了。FFmpeg 的编译是一个比较耗时的过程，而且必须先完成上面 3 项准备工作，以免出现各种错误。编译效果如图 6-2 所示。

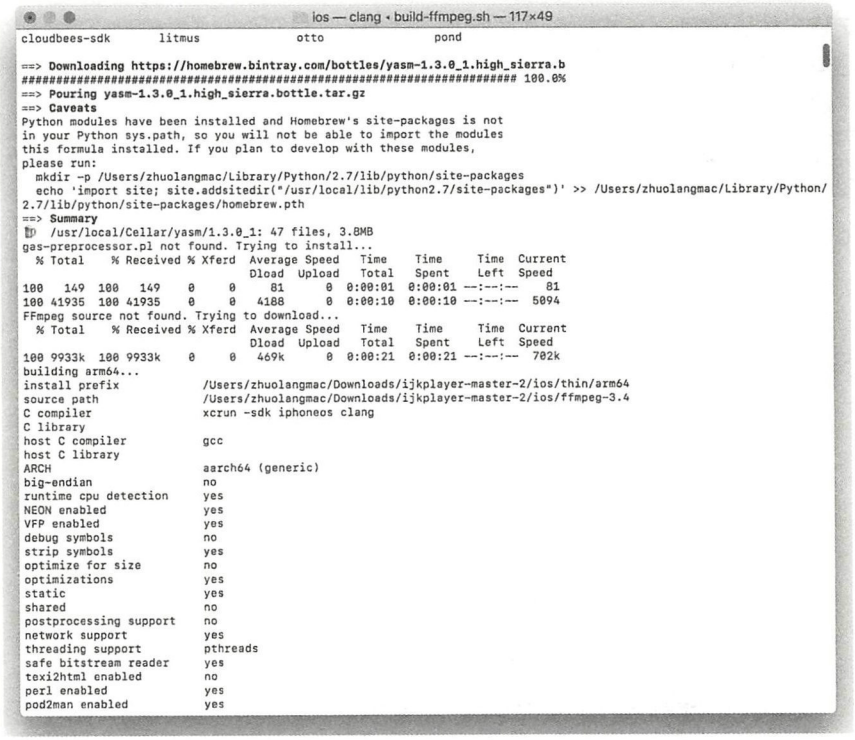


图 6-2

编译成功后的文件夹如图 6-3 所示。

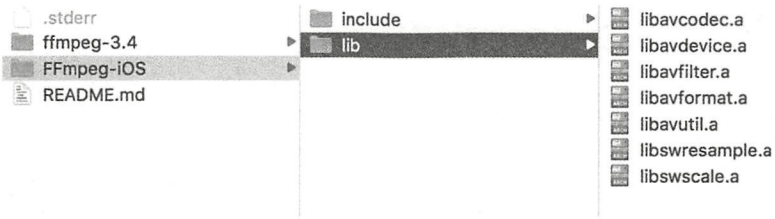


图 6-3

6.3　GPUImage 框架介绍

GPUImage 是由 Brad Larson 创建的，它是基于 GPU 的图像处理库。GPUImage 封装了 OpenGL ES 的复杂代码，并用极其简单的接口以很快的速度处理图像。

1. GPUImage 处理画面原理

GPUImage 采用链式方式来处理画面，通过 addTarget:方法为链条添加每个环节的对象，处理完一个 target，就把上一个环节处理好的图像数据传递给下一个 target 处理，这被称为 GPUImage 处理链。

2. GPUImage 的集成

使用 GPUImage 需要事先在项目中添加库：CoreMedia、CoreVideo、OpenGLES、AVFoundation、QuartzCore。导入 GPUImage 有两种方式：

（1）直接从 https://Github 中下载，然后手动导入。

（2）使用 CocoaPods 导入。

3. GPUImage 的基础类

由于 GPUImage 的功能相当全面，所以不是所有的类都会用到。下面简单介绍一下 GPUImage 的每个基础类的功能。在视频直播中，用得比较多的功能就是美颜和滤镜，其中主要用到 GPUImageFilter 和 GPUImageVideoCamera 等类。

```
#import "GLProgram.h"

// Base classes
#import "GPUImageOpenGLESContext.h"
#import "GPUImageOutput.h"
#import "GPUImageView.h"
#import "GPUImageVideoCamera.h"
#import "GPUImageStillCamera.h"
```

```
#import "GPUImageMovie.h"
#import "GPUImagePicture.h"
#import "GPUImageRawDataInput.h"
#import "GPUImageRawDataOutput.h"
#import "GPUImageMovieWriter.h"
#import "GPUImageFilterPipeline.h"
#import "GPUImageTextureOutput.h"
#import "GPUImageFilterGroup.h"
#import "GPUImageTextureInput.h"
#import "GPUImageUIElement.h"
#import "GPUImageBuffer.h"

// Filters
#import "GPUImageFilter.h"
#import "GPUImageTwoInputFilter.h"


#pragma mark - 调整颜色 Handle Color

#import "GPUImageBrightnessFilter.h"                    //亮度
#import "GPUImageExposureFilter.h"                      //曝光
#import "GPUImageContrastFilter.h"                      //对比度
#import "GPUImageSaturationFilter.h"                    //饱和度
#import "GPUImageGammaFilter.h"                         //伽马射线
#import "GPUImageColorInvertFilter.h"                   //反色
#import "GPUImageSepiaFilter.h"                         //褐色（怀旧）
#import "GPUImageLevelsFilter.h"                        //色阶
#import "GPUImageGrayscaleFilter.h"                     //灰度
#import "GPUImageHistogramFilter.h"                     //色彩直方图，显示在图片上
#import "GPUImageHistogramGenerator.h"                  //色彩直方图
#import "GPUImageRGBFilter.h"                           //RGB
#import "GPUImageToneCurveFilter.h"                     //色调曲线
#import "GPUImageMonochromeFilter.h"                    //单色
#import "GPUImageOpacityFilter.h"                       //不透明度
#import "GPUImageHighlightShadowFilter.h"               //提亮阴影
#import "GPUImageFalseColorFilter.h"                    //色彩替换（替换亮部和暗部色彩）
#import "GPUImageHueFilter.h"                           //色度
#import "GPUImageChromaKeyFilter.h"                     //色度键
#import "GPUImageWhiteBalanceFilter.h"                  //白平衡
#import "GPUImageAverageColor.h"                        //像素平均色值
#import "GPUImageSolidColorGenerator.h"                 //纯色
#import "GPUImageLuminosity.h"                          //亮度平均
#import "GPUImageAverageLuminanceThresholdFilter.h"     //像素色值亮度平均，图像黑白（有类
似漫画效果）

#import "GPUImageLookupFilter.h"                        //lookup 色彩调整
#import "GPUImageAmatorkaFilter.h"                      //Amatorka lookup
```

```
    #import "GPUImageMissEtikateFilter.h"                    //MissEtikate lookup
    #import "GPUImageSoftEleganceFilter.h"                   //SoftElegance lookup


    #pragma mark - 图像处理 Handle Image

    #import "GPUImageCrosshairGenerator.h"                   //十字形
    #import "GPUImageLineGenerator.h"                        //线条

    #import "GPUImageTransformFilter.h"                      //形状变化
    #import "GPUImageCropFilter.h"                           //剪裁
    #import "GPUImageSharpenFilter.h"                        //锐化
    #import "GPUImageUnsharpMaskFilter.h"                    //反遮罩锐化

    #import "GPUImageFastBlurFilter.h"                       //模糊
    #import "GPUImageGaussianBlurFilter.h"                   //高斯模糊
    #import "GPUImageGaussianSelectiveBlurFilter.h"          //高斯模糊，选择部分清晰
    #import "GPUImageBoxBlurFilter.h"                        //盒状模糊
    #import "GPUImageTiltShiftFilter.h"                      //条纹模糊，中间清晰，上下两端模糊
    #import "GPUImageMedianFilter.h"                         //中间值，有一种稍微模糊边缘的效果
    #import "GPUImageBilateralFilter.h"                      //双边模糊
    #import "GPUImageErosionFilter.h"                        //侵蚀边缘模糊，变黑白
    #import "GPUImageRGBErosionFilter.h"                     //RGB 侵蚀边缘模糊，有色彩
    #import "GPUImageDilationFilter.h"                       //扩展边缘模糊，变黑白
    #import "GPUImageRGBDilationFilter.h"                    //RGB 扩展边缘模糊，有色彩
    #import "GPUImageOpeningFilter.h"                        //黑白色调模糊
    #import "GPUImageRGBOpeningFilter.h"                     //彩色模糊
    #import "GPUImageClosingFilter.h"                        //黑白色调模糊，暗色会被提亮
    #import "GPUImageRGBClosingFilter.h"                     //彩色模糊，暗色会被提亮
    #import "GPUImageLanczosResamplingFilter.h"              //Lanczos 重取样，模糊效果
    #import "GPUImageNonMaximumSuppressionFilter.h"          //非最大抑制，只显示亮度最高的像素，
其他为黑色
    #import "GPUImageThresholdedNonMaximumSuppressionFilter.h" //与上相比，像素丢失更多

    #import "GPUImageSobelEdgeDetectionFilter.h"             //Sobel 边缘检测算法(白边，黑内容，
有点漫画的反色效果)
    #import "GPUImageCannyEdgeDetectionFilter.h"             //Canny 边缘检测算法（比上面更强烈
的黑白对比度）
    #import "GPUImageThresholdEdgeDetectionFilter.h"         //阈值边缘检测(效果与上面差别不大)
    #import "GPUImagePrewittEdgeDetectionFilter.h"           //Prewitt 边缘检测(效果与 Sobel
差不多，貌似更平滑)
    #import "GPUImageXYDerivativeFilter.h"                   //XYDerivative 边缘检测，画面以蓝
色为主，绿色为边缘，带彩色
    #import "GPUImageHarrisCornerDetectionFilter.h"          //Harris 角点检测，会有绿色小的“十”
形状显示在图片角点处
    #import "GPUImageNobleCornerDetectionFilter.h"           //Noble 角点检测，检测点更多
    #import "GPUImageShiTomasiFeatureDetectionFilter.h"      //ShiTomasi 角点检测，与前面差别
```

```
不大
    #import "GPUImageMotionDetector.h"                  //动作检测
    #import "GPUImageHoughTransformLineDetector.h"      //线条检测
    #import "GPUImageParallelCoordinateLineTransformFilter.h"     //平行线检测

    #import "GPUImageLocalBinaryPatternFilter.h"        //图像黑白化，并有大量噪点

    #import "GPUImageLowPassFilter.h"                   //用于图像加亮
    #import "GPUImageHighPassFilter.h"                  //图像低于某值时显示为黑


    #pragma mark - 视觉效果 Visual Effect

    #import "GPUImageSketchFilter.h"                    //素描
    #import "GPUImageThresholdSketchFilter.h"           //阈值素描，形成有噪点的素描
    #import "GPUImageToonFilter.h"                      //卡通效果（黑色粗线描边）
    #import "GPUImageSmoothToonFilter.h"                //相比上面的效果更细腻
    #import "GPUImageKuwaharaFilter.h"                  //Kuwahara 滤波，水粉画的模糊效果；处理
时间比较长，慎用

    #import "GPUImageMosaicFilter.h"                    //黑白马赛克
    #import "GPUImagePixellateFilter.h"                 //像素化
    #import "GPUImagePolarPixellateFilter.h"            //同心圆像素化
    #import "GPUImageCrosshatchFilter.h"                //交叉线阴影，形成黑白网状画面
    #import "GPUImageColorPackingFilter.h"              //色彩丢失，模糊（类似监控摄像效果）

    #import "GPUImageVignetteFilter.h"              //晕影，形成黑色圆形边缘，突出中间图像的效果
    #import "GPUImageSwirlFilter.h"                     //漩涡，中间形成卷曲的画面
    #import "GPUImageBulgeDistortionFilter.h"           //鱼眼效果
    #import "GPUImagePinchDistortionFilter.h"           //收缩失真，凹面镜效果
    #import "GPUImageStretchDistortionFilter.h"         //伸展失真，哈哈镜效果
    #import "GPUImageGlassSphereFilter.h"               //水晶球效果
    #import "GPUImageSphereRefractionFilter.h"          //球形折射，图形倒立效果

    #import "GPUImagePosterizeFilter.h"                 //色调分离，形成噪点效果
    #import "GPUImageCGAColorspaceFilter.h"           //CGA 色彩滤镜，形成黑、浅蓝、紫色块的画面
    #import "GPUImagePerlinNoiseFilter.h"               //柏林噪点，花边噪点
    #import "GPUImage3x3ConvolutionFilter.h" //3×3 卷积，高亮、大色块、变黑，加亮边缘、线条等
    #import "GPUImageEmbossFilter.h"                    //浮雕效果，带有三维的感觉
    #import "GPUImagePolkaDotFilter.h"                  //像素圆点花样
    #import "GPUImageHalftoneFilter.h"              //点染，图像黑白化，由黑点构成原图的大致图形


    #pragma mark - 混合模式 Blend

    #import "GPUImageMultiplyBlendFilter.h"             //通常用于创建阴影和深度效果
    #import "GPUImageNormalBlendFilter.h"               //正常
```

```
    #import "GPUImageAlphaBlendFilter.h"          //透明混合，通常用于在背景上应用前景的透明度
    #import "GPUImageDissolveBlendFilter.h"          //溶解
    #import "GPUImageOverlayBlendFilter.h"           //叠加，通常用于创建阴影效果
    #import "GPUImageDarkenBlendFilter.h"            //加深混合，通常用于重叠类型
    #import "GPUImageLightenBlendFilter.h"           //减淡混合，通常用于重叠类型
    #import "GPUImageSourceOverBlendFilter.h"        //源混合
    #import "GPUImageColorBurnBlendFilter.h"         //色彩加深混合
    #import "GPUImageColorDodgeBlendFilter.h"        //色彩减淡混合
    #import "GPUImageScreenBlendFilter.h"            //屏幕包裹，通常用于创建亮点和镜头眩光
    #import "GPUImageExclusionBlendFilter.h"         //排除混合
    #import "GPUImageDifferenceBlendFilter.h"        //差异混合，通常用于创建更多变动的颜色
    #import "GPUImageSubtractBlendFilter.h"          //差值混合，通常用于创建两个图像之间的动
画变暗模糊效果
    #import "GPUImageHardLightBlendFilter.h"         //强光混合，通常用于创建阴影效果
    #import "GPUImageSoftLightBlendFilter.h"         //柔光混合
    #import "GPUImageChromaKeyBlendFilter.h"         //色度键混合
    #import "GPUImageMaskFilter.h"                   //遮罩混合
    #import "GPUImageHazeFilter.h"                   //朦胧加暗
    #import "GPUImageLuminanceThresholdFilter.h"     //亮度阈
    #import "GPUImageAdaptiveThresholdFilter.h"      //自适应阈值
    #import "GPUImageAddBlendFilter.h"           //通常用于创建两个图像之间的动画变亮模糊效果
    #import "GPUImageDivideBlendFilter.h"        //通常用于创建两个图像之间的动画变暗模糊效果

    #import "GPUImageJFAVoroniFilter.h"
    #import "GPUImageVoroniConsumerFilter.h"
```

6.4　LFLiveKit 框架介绍

LFLiveKit 框架是实现直播推流的开源框架，其利用 H.264 和 AAC 硬编码，支持 GPUImage 美化、RTMP 传输推流、弱网络丢帧，以及支持动态切换码率功能。

1. LFLiveKit 的集成

（1）在导入 LFLiveKit 之前，需要先导入一些相关类库：UIKit、Foundation、AVFoundation、VideoToolbox、AudioToolbox、libz、libstdc++。

（2）导入这些相关库依然有两种方式：

- 直接从 GitHub 上下载 LFLivekit，然后手动导入。
- 使用 CocoaPods 导入。

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '7.0'
pod 'LFLiveKit'
```

（3）引入头文件：

```
#Import <LFLiveKit/LFLiveKit.h>
```

注意：如果导入 LFLiveKit，则无须再导入 GPUImage，因为 LFLiveKit 已经为我们集成了 GPUImage，如图 6-4 所示。

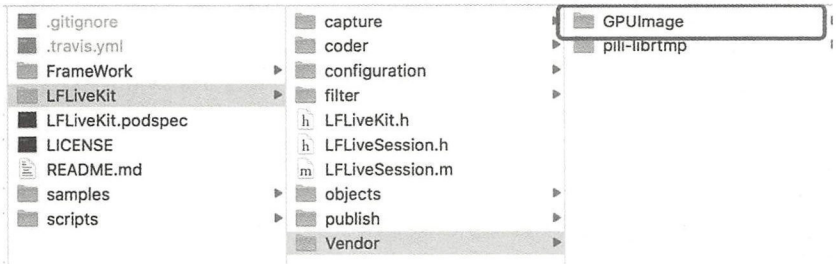


图 6-4

2. LFLiveKit 的基础架构

LFLiveKit 的基础架构如表 6-1 所示。

表 6-1

类　名	说　明
LFLiveSession	它是整个 SDK 核心，提供对外部的主要接口。其主要功能有：管理推流开关，管理音/视频的录制及渲染，管理录制渲染后的音/视频编码，管理编码后的数据上传，管理音/视频的基础配置，回调推流状态和上报异常等
LFLiveAudioConfiguration	音频配置类，负责配置相关音频信息（音频质量、码率、采样率、声道数）
LFLiveVideoConfiguration	视频配置类，负责配置相关音频基本信息（视频质量、码率、帧数、分辨率）和应用配置，如最大、最小帧率等
LFVideoCapture	视频管理类，负责管理视频的输入和输出。同时处理业务需求，如美颜、亮度、添加水印等效果。它用了一个第三方工具——GPUIImage 处理渲染效果
LFAudioCapture	音频管理类，负责管理音频的输入开关。这里没有多余限制，用原生的 API 即可
LFH264VideoEncoder, LFHardwareVideoEncoder	视频编码类，分别对应 8.0 以前和 8.0 以后的两种设备的视频编码类。都遵守 LFVideoEncoding 协议，并设置 LFStreamSocketDelegate 协议给 session 管理
LFHardwareAudioEncoder	音频编码类，遵守 LFVideoEncoding 协议，并设置 LFStreamSocketDelegate 协议给 session 管理
LFFrame	数据信息的基类，作为上传到服务器数据的基本模型
LFVideoFrame	视频信息，作为上传到服务器视频数据的模型
LFAudioFrame	音频信息，作为上传到服务器音频数据的模型

续表

类　名	说　明
LFLiveStreamInfo	推流信息，包括推流地址（目前主要应用 rtmp 推流），流状态，音/视频配置信息，异常信息
LFStreamRTMPSocket	数据上传管理类，负责开/关数据上传，回调连接状态和异常。遵循 LFStreamSocket 协议，并设置 LFStreamSocketDelegate 给 session 管理
LFLiveDebug	调试信息，其是开发时候的内部表示，主要用于记录调试作用
LFStreamingBuffer	本地采样，通过本地采样监控缓冲区，可实现相关切换帧率码率等策略

3. LFLiveKit 的简单使用

（1）Objective-C。

```
- (LFLiveSession*)session {
    if (!_session) {
        _session = [[LFLiveSession alloc]
initWithAudioConfiguration:[LFLiveAudioConfiguration defaultConfiguration]
videoConfiguration:[LFLiveVideoConfiguration defaultConfiguration]];
        _session.preView = self;
        _session.delegate = self;
    }
    return _session;
}

- (void)startLive {
    LFLiveStreamInfo *streamInfo = [LFLiveStreamInfo new];
    streamInfo.url = @"your server rtmp url";
    [self.session startLive:streamInfo];
}

- (void)stopLive {
    [self.session stopLive];
}

//MARK: - CallBack:
- (void)liveSession:(nullable LFLiveSession *)session liveStateDidChange:
(LFLiveState)state;
- (void)liveSession:(nullable LFLiveSession *)session debugInfo:(nullable
LFLiveDebug*)debugInfo;
- (void)liveSession:(nullable LFLiveSession*)session
errorCode:(LFLiveSocketErrorCode)errorCode;
```

（2）Swift。

```
// import LFLiveKit in [ProjectName]-Bridging-Header.h
#import <LFLiveKit.h>
```

```
    //MARK: - Getters and Setters
    lazy var session: LFLiveSession = {
        let audioConfiguration = LFLiveAudioConfiguration.defaultConfiguration()
        let videoConfiguration =
LFLiveVideoConfiguration.defaultConfigurationForQuality(LFLiveVideoQuality.Low3,
landscape: false)
        let session = LFLiveSession(audioConfiguration: audioConfiguration,
videoConfiguration: videoConfiguration)

        session?.delegate = self
        session?.preView = self.view
        return session!
    }()

    //MARK: - Event
    func startLive() -> Void {
        let stream = LFLiveStreamInfo()
        stream.url = "your server rtmp url";
        session.startLive(stream)
    }

    func stopLive() -> Void {
        session.stopLive()
    }

    //MARK: - Callback
    func liveSession(session: LFLiveSession?, debugInfo: LFLiveDebug?)
    func liveSession(session: LFLiveSession?, errorCode: LFLiveSocketErrorCode)
    func liveSession(session: LFLiveSession?, liveStateDidChange state: LFLiveState)
```

6.5 IJKPlayer 框架介绍

IJKPlayer 是一个基于 FFmpeg 的轻量级 Android/iOS 端视频播放器。其实现了跨平台功能，具有 API 易于集成、编译配置可裁剪、支持硬件加速解码、更加省电等优点。目前比较火的美拍和斗鱼 App 都在使用这个框架。

1. IJKPlayer 的编译运行

（1）首先下载 IJKPlayer 安装文件。

（2）在解压缩后的 IJKPlayer 文件夹中，找到 IJKMediaDemo 文件。打开 IJKMediaDemo 文件，编译会提示“libavformat/avformat.h'file not found”的错误，这是因为 libavformat 是 FFmpeg 中的库，而 IJKPlayer 是基于 FFmpeg 这个库的，因此需要导入 FFmpeg。

（3）使用终端，打开 IJKPlayer-master 目录，输入./init-ios.sh 命令运行脚本文件。init-ios.sh 命令的作用：下载 FFmpeg 源码，如果在前期准备中已经编译了 FFmpeg，则可以直接到 FFmpeg 中导入需要的文件。

（4）执行完脚本后，就会发现 IJKPlayer 中有 FFmpeg 了，但这仅仅是把 FFmpeg 下载了，并没有编译，我们还要继续编译。

在编译时需要在终端输入如下命令：

```
cd ios
./compile-ffmpeg.sh clean
./compile-ffmpeg.sh all
```

./compile-ffmpeg.sh clean 用于删除一些文件和文件夹，为编译 ffmpeg.sh 做准备。./compile-ffmpeg.sh all 是真正的编译各个平台的 FFmpeg 库，并生成所有平台的通用库。如图 6-5 所示，此时 FFmpeg 已经彻底编译成功。

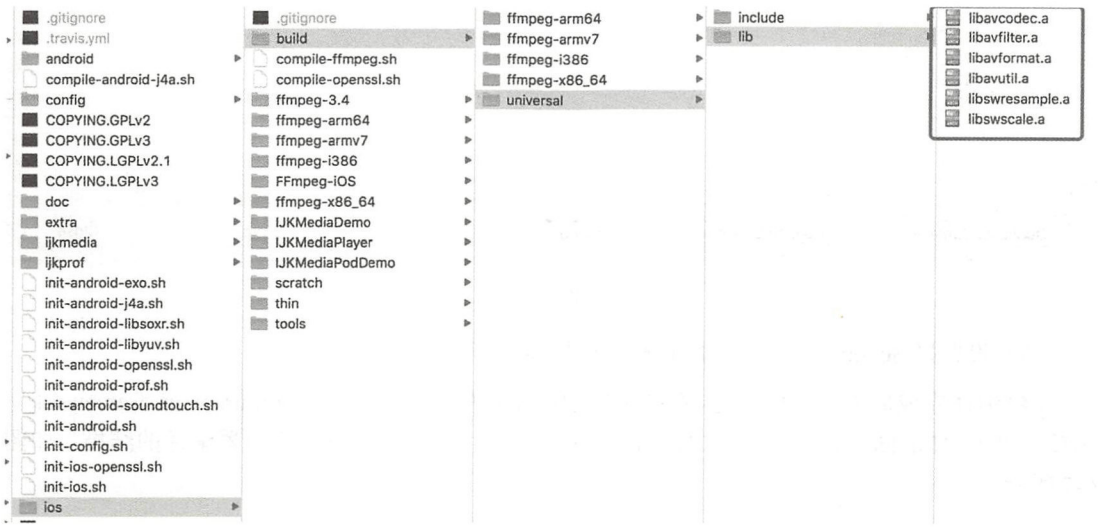


图 6-5

（5）可以运行代码了。

2. IJKPlayer 的集成

IJKPlayer 的集成有两种方法：一种方法是像在 IJKMediaDemo 工程中那样，直接导入工程 IJKMediaPlayer.xcodeproj，这种方法比较初级，这里就不多做介绍了。第二种方法是把 IJKPlayer 打包成 Framework，然后导入工程中使用。这种方法比较利于后期移植，方便使用。下面介绍如何把 IJKPlayer 打成包成 Framework。

（1）首先打开工程 IJKMediaPlayer.xcodeproj。

（2）单击“Edit Scheme”按钮，在打开的对话框中设置 Scheme，将“Build Configuration”改成“Release”，如图 6-6 所示。

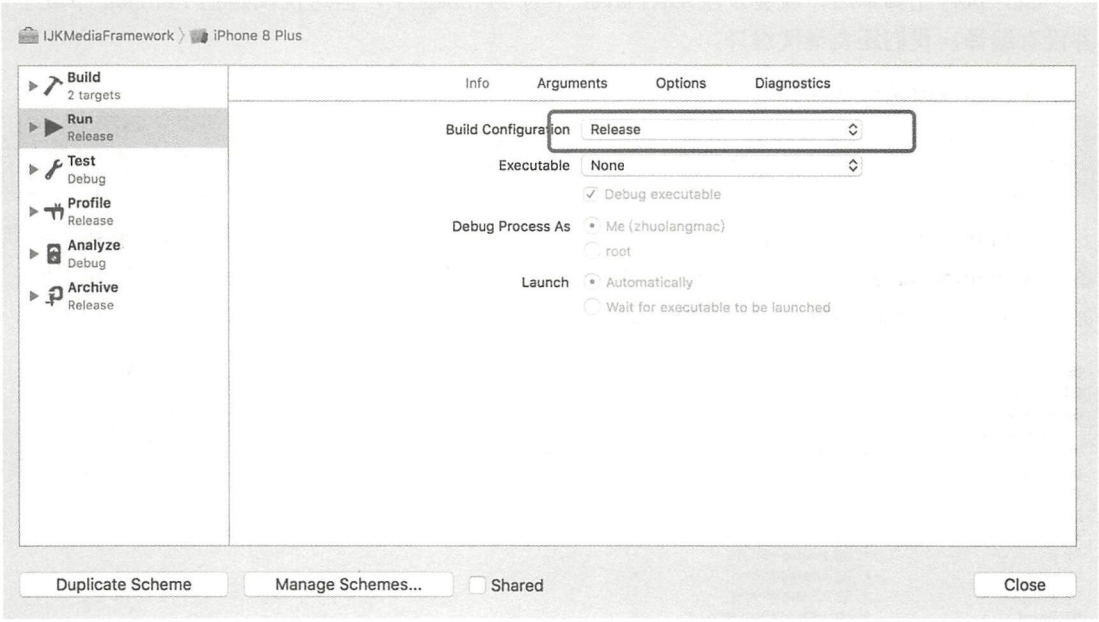


图 6-6

（3）设置好 Scheme 后，分别用真机和模拟器进行编译。

（4）编译完成后单击鼠标右键，在弹出的快捷菜单中选择“IJKMediaFramework.framework”选项，进入“Finder”窗口。进入“Finder”窗口后，可以看到真机和模拟器编译的结果，如图 6-7 所示。

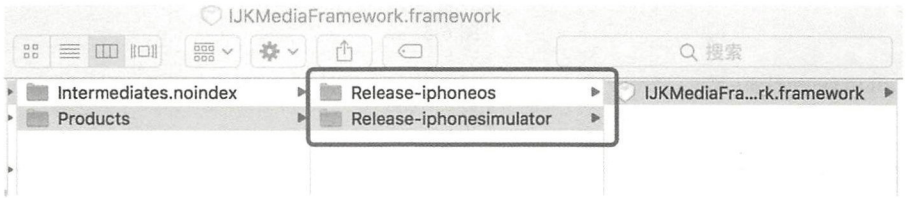


图 6-7

（5）打开终端并合并两个版本的 Framework，输入命令行，命令行格式如下：

```
    lipo -create "真机版本 framework 路径""模拟器版本 framework 路径" -output "合并后的文件路径
"
```

注意：我们合并的 Framework 是/Release-iphoneos/IJKMediaFramework.framework /IJKMedia Framework 和/Release-iphonesimulator/IJKMediaFramework.framework/ IJKMediaFramework 这两个文件。

（6）合并成功后，需要用合并完的 IJKMediaFramework 替换原来其中一个版本的 IJKMediaFramework，生成新的 Framework，如图 6-8 所示。

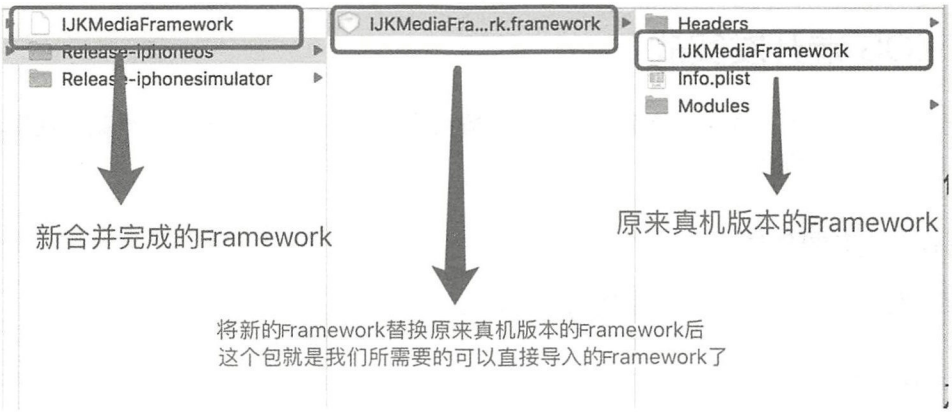


图 6-8

（7）至此，替换过新的 Framework 后的 IJKMediaFramework.framework 文件就是我们需要的框架了，可以将其复制，再导入我们的工程中并使用。在本书中也会提供制作好的 IJKMediaFramework.framework。

（8）在工程中，除要导入新的 IJKMediaFramework.framework 外，还要导入别的依赖库：libz.tbd、libbz2.tbd、libstdc++.tbd、AudioToolbox.framework、AVFoundation.framework CoreGraphics.framework、CoreMedia.framework、MediaPlayer.framework MobileCoreServices.framework、OpenGLES.framework、QuartzCore.framework UIKit.framework VideoToolbox.framework。

3. IJKplayer 的简单使用

```
IJKFFOptions *options = [IJKFFOptions optionsByDefault];
[options setPlayerOptionIntValue:1  forKey:@"videotoolbox"];
// 帧速率(fps)（可以改，确认非标准帧速率会导致音画不同步，所以其值只能设定为15fps或者29.97fps）
[options setPlayerOptionIntValue:29.97 forKey:@"r"];
// -vol——设置音量大小，256为标准音量
[options setPlayerOptionIntValue:512 forKey:@"vol"];
IJKFFMoviePlayerController *moviePlayer = [[IJKFFMoviePlayerController alloc]
```

```
initWithContentURLString:flv withOptions:options];
        moviePlayer.view.frame = self.contentView.bounds;
        // 填充 fill
        moviePlayer.scalingMode = IJKMPMovieScalingModeAspectFill;
        // 设置自动播放(必须设置为 NO，防止自动播放，才能更好地控制直播的状态)
        moviePlayer.shouldAutoplay = NO;
        // 默认不显示
        moviePlayer.shouldShowHudView = NO;
        [self.contentView insertSubview:moviePlayer.view atIndex:0];
        [moviePlayer prepareToPlay];
        self.moviePlayer = moviePlayer;
```

6.6 iOS 端开发实战

通过前面的学习，读者应该掌握了开发直播系统需要的基础知识和基本框架的使用。本节介绍如何开发一个直播 App。

6.6.1 主要功能

要开发一个直播 App，主要功能包括以下几种。

1. 直播端

- 设置美颜。
- 切换前/后摄像头。
- 开始/结束推流。
- 调整画质。

2. 播放端

- 开始/结束拉流。
- 调整画质。
- 点赞动画。
- 开启/关闭弹幕。

3. 实现效果

- 直播端（见图 6-9）。
- 播放端（见图 6-10）。

图 6-9

图 6-10

6.6.2　框架导入

然后创建新工程，导入所需要的框架和依赖库，如图 6-11 所示。

Name	Status
IJKMediaFramework.framework	Required
QuartzCore.framework	Required
OpenGLES.framework	Required
MobileCoreServices.framework	Required
MediaPlayer.framework	Required
CoreMedia.framework	Required
CoreGraphics.framework	Required
libbz2.tbd	Required
libstdc++.tbd	Required
libz.tbd	Required
AudioToolbox.framework	Required
VideoToolbox.framework	Required
AVFoundation.framework	Required
Foundation.framework	Required
UIKit.framework	Required

图 6-11

6.6.3 滤镜

在开启直播前，通常会对采集到的视频进行美化处理，使场景看上去更美观。常用的视频美化处理包括磨皮、美白、提高亮度、提高饱和度等，这些操作都是滤镜处理的一种。

滤镜处理的原理是：把静态图片或者视频的每一帧进行图形变换再显示出来。其本质就是变换像素点的坐标和颜色。要实现滤镜功能，就需要使用 GPUImage 框架，也可以直接使用 LFLiveKit 框架，LFLiveKit 框架内部已经对 GPUImage 做了一次简单的封装。本节会分别介绍这两种框架在视频美化上的应用。

1. LFLiveKit 框架

LFLiveKit 框架中的 LFGPUImageBeautyFilter 就是已经封装好 GPUImage 的类，可以直接调用里面的方法进行美颜。这个类提供了 3 种方法，分别用于实现美白程度调节、亮度调节、色调调节，代码如下：

```
    - (id)init;
    {
        if (!(self = [super
initWithFragmentShaderFromString:kLFGPUImageBeautyFragmentShaderString])) {
            return nil;
        }
        //设置色彩饱和度
        _toneLevel = 0.5;
        //设置美白程度
        _beautyLevel = 0.5;
        //设置亮度
        _brightLevel = 0.5;
        [self setParams:_beautyLevel tone:_toneLevel];
        [self setBrightLevel:_brightLevel];
        return self;
    }

    - (void)setInputSize:(CGSize)newSize atIndex:(NSInteger)textureIndex {
        [super setInputSize:newSize atIndex:textureIndex];
        inputTextureSize = newSize;

        CGPoint offset = CGPointMake(2.0f / inputTextureSize.width, 2.0 /
inputTextureSize.height);
        [self setPoint:offset forUniformName:@"singleStepOffset"];
    }
    //调节美颜等级
    - (void)setBeautyLevel:(CGFloat)beautyLevel {
        _beautyLevel = beautyLevel;
        [self setParams:_beautyLevel tone:_toneLevel];
```

```
}
//调节亮度
- (void)setBrightLevel:(CGFloat)brightLevel {
    _brightLevel = brightLevel;
    [self setFloat:0.6 * (-0.5 + brightLevel) forUniformName:@"brightness"];
}

- (void)setParams:(CGFloat)beauty tone:(CGFloat)tone {
    GPUVector4 fBeautyParam;
    fBeautyParam.one = 1.0 - 0.6 * beauty;
    fBeautyParam.two = 1.0 - 0.3 * beauty;
    fBeautyParam.three = 0.1 + 0.3 * tone;
    fBeautyParam.four = 0.1 + 0.3 * tone;
    [self setFloatVec4:fBeautyParam forUniform:@"params"];
}
```

这些功能使用起来非常方便，直接调用即可，代码如下：

```
//开启或关闭美颜功能
self.session.beautyFace=YES;
//设置美颜程度
self.session.beautyLevel=0.5;
//设置亮度
self.session.brightLevel=0.5;
```

2. GPUImage 框架

在 6.3 节中讲过 GPUImage 框架是基于 OpenGL ES 的，而使用 OpenGL ES 用来处理图片，一般会分为 4 步：

（1）初始化 OpenGL ES 环境，编译、连接顶点着色器和片元着色器。

（2）缓存顶点、纹理坐标数据，传送图像数据到 GPU。

（3）绘制图元到特定的帧缓存中。

（4）在帧缓存中取出绘制的图像。

在 GPUImage 框架中，GPUImageFilter 类主要负责上述第（1）~（3）步，GPUImageFramebufferz 类主要负责第（4）步。了解了 OpenGL ES 的处理过程，再来看一看 GPUImage 框架的处理过程。在 6.3 节中讲过 GPUImage 采用链式方式来处理画面，具体分为 3 个环节：source→filter→final target。

（1）source（视频、图片源）。

GPUImageVideoCamera：用于实时拍摄视频。

GPUImageStillCamera：用于实时拍摄照片。

GPUImagePicture：用于处理已经拍摄好的图片。

GPUImageMovie：用于处理已经拍摄好的视频。

（2）filter（滤镜）。

GPUImageFilter：用来接收源图像，通过自定义的顶点、片元着色器来渲染新的图像，并在绘制完成后通知响应链的下一个对象。

GPUImageFramebuffer：用来管理纹理缓存的格式与读/写帧缓存的 buffer。

（3）final target（处理后的视频、图片）。

GPUImageView 和 GPUImageMovieWriter：最终输入目标，显示图片或者视频。

美颜功能主要实现两个效果：磨皮和美白。

- 磨皮（GPUImageBilateralFilter）：本质就是让像素点模糊。可以使用高斯模糊，但是可能会导致图像边缘不清晰。而使用双边滤波（Bilateral Filter），可以有针对性地模糊像素点，能保证边缘不被模糊。
- 美白（GPUImageBrightnessFilter）：本质就是提高亮度。

将这两个功能组合成滤镜组链使用，就会实现我们想要的美颜效果。实现代码如下：

```
    - (void)viewDidLoad {
        [super viewDidLoad];
        // 创建视频源
        // SessionPreset:屏幕分辨率，AVCaptureSessionPresetHigh 会自适应高分辨率
        // cameraPosition:摄像头方向
        GPUImageVideoCamera *videoCamera = [[GPUImageVideoCamera alloc]
initWithSessionPreset:AVCaptureSessionPresetHigh
cameraPosition:AVCaptureDevicePositionFront];
         videoCamera.outputImageOrientation = UIInterfaceOrientationPortrait;
        _videoCamera = videoCamera;

        // 创建最终输出 View
        GPUImageView *captureVideoPreview = [[GPUImageView alloc]
initWithFrame:self.view.bounds];
        [self.view insertSubview:captureVideoPreview atIndex:0];

        // 创建滤镜：磨皮和美白，这个就是我们组合滤镜，被称为滤镜组
        GPUImageFilterGroup *groupFilter = [[GPUImageFilterGroup alloc] init];

        // 磨皮滤镜
        GPUImageBilateralFilter *bilateralFilter = [[GPUImageBilateralFilter alloc]
init];
        [groupFilter addTarget:bilateralFilter];
        _bilateralFilter = bilateralFilter;
```

```
        // 美白滤镜
        GPUImageBrightnessFilter *brightnessFilter = [[GPUImageBrightnessFilter alloc]
init];
        [groupFilter addTarget:brightnessFilter];
        _brightnessFilter = brightnessFilter;

        // 设置滤镜组链
        [bilateralFilter addTarget:brightnessFilter];
        [groupFilter setInitialFilters:@[bilateralFilter]];
        groupFilter.terminalFilter = brightnessFilter;

        // 设置 GPUImage 响应链，从数据源→滤镜组→最终界面效果
        [videoCamera addTarget:groupFilter];
        [groupFilter addTarget:captureVideoPreview];

        // 必须调用 startCameraCapture，底层把采集到的视频源渲染到 GPUImageView 中，就可以显示了
        // 开始采集视频
        [videoCamera startCameraCapture];
    }

    - (IBAction)brightnessFilter:(UISlider *)sender {
        _brightnessFilter.brightness = sender.value;
    }

    - (IBAction)bilateralFilter:(UISlider *)sender {
        // 值越小，磨皮效果越好
        CGFloat maxValue = 10;
        [_bilateralFilter setDistanceNormalizationFactor:(maxValue - sender.value)];
    }
```

注意：

- SessionPreset 最好使用 AVCaptureSessionPresetHigh，会自动识别，如果分辨率太高，则设备不支持时会直接报错。
- GPUImageVideoCamera 必须要强引用，否则会被销毁，不能持续采集视频。
- 必须调用 startCameraCapture，底层才会把采集到的视频源渲染到 GPUImageView 中。
- GPUImageBilateralFilter 的 distanceNormalizationFactor 值越小，磨皮效果越好，distanceNormalizationFactor 取值要大于 1。

GPUImage 是相当强大的开源框架，它能实现的效果远远比美颜效果多得多，它的源码也有很好的学习价值。

6.6.4 推流

这里使用 LFLiveKit 框架中的 LFLiveSession 进行推流。LFLiveSession 常用属性如表 6-2 所示。

表 6-2

属　性	类　型	说　明
running	BOOL	控制直播开始/结束
preView	UIView	视频图层
captureDevicePosition	AVCaptureDevicePosition	摄像头方向
beautyFace	BOOL	美颜开关
muted	BOOL	静音开关
streamInfo	LFLiveStreamInfo	控制直播流信息
state	LFLiveState	直播流上传状态
showDebugInfo	BOOL	是否显示调试信息
reconnectInterval	NSUInteger	重连间隔

1. 实现步骤

（1）引用头文件和 LFLiveSessionDelegate 代理。

```
#import "LFLiveKit.h"

@interface PushFlowViewController ()<LFLiveSessionDelegate>
```

（2）创建变量。

```
@property (nonatomic, strong) LFLiveDebug *debugInfo;
@property (nonatomic, strong) LFLiveSession *session;
```

（3）初始化 session。

```
_session = [[LFLiveSession alloc]
initWithAudioConfiguration:[LFLiveAudioConfiguration defaultConfiguration]
videoConfiguration:videoConfiguration captureType:LFLiveCaptureDefaultMask];
_session.delegate = self;
_session.showDebugInfo = NO;
_session.preView = self.view;
```

（4）设置 LFLiveVideoConfiguration。

```
/*** 默认分辨率为 368px × 640px 音频：44.1KB， iPhone6 以上 48KB 双声道方向竖屏 ***/
LFLiveVideoConfiguration *videoConfiguration = [LFLiveVideoConfiguration
new];
videoConfiguration.videoSize = CGSizeMake(360, 640);
videoConfiguration.videoBitRate = 800*1024;
videoConfiguration.videoMaxBitRate = 1000*1024;
```

```
            videoConfiguration.videoMinBitRate = 500*1024;
            videoConfiguration.videoFrameRate = 24;
            videoConfiguration.videoMaxKeyframeInterval = 48;
            videoConfiguration.outputImageOrientation = UIInterfaceOrientationPortrait;
            videoConfiguration.autorotate = NO;
            videoConfiguration.sessionPreset = LFCaptureSessionPreset720x1280;
            _session = [[LFLiveSession alloc]
initWithAudioConfiguration:[LFLiveAudioConfiguration defaultConfiguration]
videoConfiguration:videoConfiguration captureType:LFLiveCaptureDefaultMask];

            /**    自己定制单声道  */
            /*
             LFLiveAudioConfiguration *audioConfiguration = [LFLiveAudioConfiguration
new];
             audioConfiguration.numberOfChannels = 1;
             audioConfiguration.audioBitrate = LFLiveAudioBitRate_64Kbps;
             audioConfiguration.audioSampleRate = LFLiveAudioSampleRate_44100Hz;
             _session = [[LFLiveSession alloc]
initWithAudioConfiguration:audioConfiguration
videoConfiguration:[LFLiveVideoConfiguration defaultConfiguration]];
             */

            /**    自己定制高质量音频为 96KB*/
            /*
             LFLiveAudioConfiguration *audioConfiguration = [LFLiveAudioConfiguration
new];
             audioConfiguration.numberOfChannels = 2;
             audioConfiguration.audioBitrate = LFLiveAudioBitRate_96Kbps;
             audioConfiguration.audioSampleRate = LFLiveAudioSampleRate_44100Hz;
             _session = [[LFLiveSession alloc]
initWithAudioConfiguration:audioConfiguration
videoConfiguration:[LFLiveVideoConfiguration defaultConfiguration]];
             */

            /**    自己定制高质量音频为 96KB，分辨率设置为 540px×960px ，方向为竖屏 */

            /*
             LFLiveAudioConfiguration *audioConfiguration = [LFLiveAudioConfiguration
new];
             audioConfiguration.numberOfChannels = 2;
             audioConfiguration.audioBitrate = LFLiveAudioBitRate_96Kbps;
             audioConfiguration.audioSampleRate = LFLiveAudioSampleRate_44100Hz;

             LFLiveVideoConfiguration *videoConfiguration = [LFLiveVideoConfiguration
new];
             videoConfiguration.videoSize = CGSizeMake(540, 960);
             videoConfiguration.videoBitRate = 800*1024;
```

```
            videoConfiguration.videoMaxBitRate = 1000*1024;
            videoConfiguration.videoMinBitRate = 500*1024;
            videoConfiguration.videoFrameRate = 24;
            videoConfiguration.videoMaxKeyframeInterval = 48;
            videoConfiguration.orientation = UIInterfaceOrientationPortrait;
            videoConfiguration.sessionPreset = LFCaptureSessionPreset540x960;

            _session = [[LFLiveSession alloc]
initWithAudioConfiguration:audioConfiguration videoConfiguration:videoConfiguration];
            */


           /**    自己定制高质量音频为 128KB，分辨率设置为 720px×1280px，方向为竖屏 */

           /*
            LFLiveAudioConfiguration *audioConfiguration = [LFLiveAudioConfiguration
new];
            audioConfiguration.numberOfChannels = 2;
            audioConfiguration.audioBitrate = LFLiveAudioBitRate_128Kbps;
            audioConfiguration.audioSampleRate = LFLiveAudioSampleRate_44100Hz;

            LFLiveVideoConfiguration *videoConfiguration = [LFLiveVideoConfiguration
new];
            videoConfiguration.videoSize = CGSizeMake(720, 1280);
            videoConfiguration.videoBitRate = 800*1024;
            videoConfiguration.videoMaxBitRate = 1000*1024;
            videoConfiguration.videoMinBitRate = 500*1024;
            videoConfiguration.videoFrameRate = 15;
            videoConfiguration.videoMaxKeyframeInterval = 30;
            videoConfiguration.landscape = NO;
            videoConfiguration.sessionPreset = LFCaptureSessionPreset360x640;

            _session = [[LFLiveSession alloc]
initWithAudioConfiguration:audioConfiguration videoConfiguration:videoConfiguration];
            */


           /**    自己定制高质量音频为 128KB，分辨率设置为 720px×1280px，方向为横屏  */

           /*
            LFLiveAudioConfiguration *audioConfiguration = [LFLiveAudioConfiguration
new];
            audioConfiguration.numberOfChannels = 2;
            audioConfiguration.audioBitrate = LFLiveAudioBitRate_128Kbps;
            audioConfiguration.audioSampleRate = LFLiveAudioSampleRate_44100Hz;

            LFLiveVideoConfiguration *videoConfiguration = [LFLiveVideoConfiguration
```


```
new];
            videoConfiguration.videoSize = CGSizeMake(1280, 720);
            videoConfiguration.videoBitRate = 800*1024;
            videoConfiguration.videoMaxBitRate = 1000*1024;
            videoConfiguration.videoMinBitRate = 500*1024;
            videoConfiguration.videoFrameRate = 15;
            videoConfiguration.videoMaxKeyframeInterval = 30;
            videoConfiguration.landscape = YES;
            videoConfiguration.sessionPreset = LFCaptureSessionPreset720x1280;

            _session = [[LFLiveSession alloc]
initWithAudioConfiguration:audioConfiguration videoConfiguration:videoConfiguration];
            */
```

（5）实现代理方法。

```
    /** live status changed will callback */
    - (void)liveSession:(nullable LFLiveSession *)session
liveStateDidChange:(LFLiveState)state {
        NSLog(@"session: %@", session);
        NSLog(@"liveStateDidChange: %ld", state);
        switch (state) {
            case LFLiveReady:
                _stateLabel.text = @"未连接";
                isRunning=NO;
                 [self.playButton setTitle:@"开始直播" forState:UIControlStateNormal];
                break;
            case LFLivePending:
                _stateLabel.text = @"连接中";
                break;
            case LFLiveStart:
                _stateLabel.text = @"已连接";
                break;
            case LFLiveError:
                _stateLabel.text = @"连接错误";
                isRunning=NO;
                 [self.playButton setTitle:@"开始直播" forState:UIControlStateNormal];
                break;
            case LFLiveStop:
                _stateLabel.text = @"已断开";
                isRunning=NO;
                 [self.playButton setTitle:@"开始直播" forState:UIControlStateNormal];
                break;
            default:
                break;
        }
    }
```


```
    /** live debug info callback */
    //直播流的信息，如果需要显示当前流量和实时码率等信息，则可以在这个方法里实现
    - (void)liveSession:(nullable LFLiveSession *)session debugInfo:(nullable
LFLiveDebug *)debugInfo {
        NSLog(@"debugInfo uploadSpeed: %@", formatedSpeed(debugInfo.currentBandwidth,
debugInfo.elapsedMilli));
    }

    /** callback socket errorcode */
    //连接失败
    - (void)liveSession:(nullable LFLiveSession *)session
errorCode:(LFLiveSocketErrorCode)errorCode {
        NSLog(@"errorCode: %ld", errorCode);
        isRunning=NO;
         [self.playButton setTitle:@"开始直播" forState:UIControlStateNormal];
    }
```

（6）开始直播。

```
    LFLiveStreamInfo *stream = [LFLiveStreamInfo new];
    stream.url=@"rtmp://172.26.1.92:1935/liveOnline/6000";
    [self.session startLive:stream];
```

（7）结束直播。

```
    [self.session stopLive];
```

2. 实现效果

实现效果如图 6-12 所示。

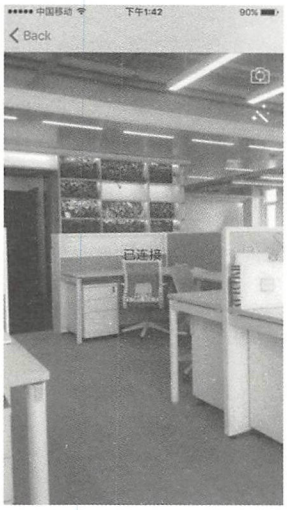


图 6-12

6.6.5　拉流

拉流使用 IJKPlayer 框架，本节介绍 IJKPlayer 框架具体的拉流过程。

3．实现步骤

（1）引入头文件。

```
#import <IJKMediaFramework/IJKMediaFramework.h>
#import <AVKit/AVKit.h>
#import <AVFoundation/AVFoundation.h>
```

（2）创建变量。

```
@property (atomic, strong) NSURL *url;
@property (atomic, retain) id <IJKMediaPlayback> player;
```

（3）初始化 player。

```
        _player = [[IJKFFMoviePlayerController alloc] initWithContentURL:self.url
withOptions:nil];

        UIView *playerView = [self.player view];
        UIView *displayView = [[UIView alloc] initWithFrame:CGRectMake(0, 64,
self.view.bounds.size.width, 500)];
        self.PlayerView = displayView;
        self.PlayerView.backgroundColor = [UIColor blackColor];
        [self.view addSubview:self.PlayerView];

        playerView.frame = self.PlayerView.bounds;
        playerView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;

        [self.PlayerView insertSubview:playerView atIndex:1];
        //缩放模式为 FILL
        [_player setScalingMode:IJKMPMovieScalingModeAspectFill];
```

（4）设置 IJKFFOptions。

- 设备参数设置。

```
      IJKFFOptions *options = [IJKFFOptions optionsByDefault]; //使用默认配置

        //开启硬解码
        [options setPlayerOptionIntValue:0 forKey:@"videotoolbox"];

        // 设置音量大小，256 为标准音量（要设置成两倍音量时，则输入 512，依此类推）
        [options setPlayerOptionIntValue:512 forKey:@"vol"];

        // 最大 fps
```


```
        [options setPlayerOptionIntValue:30 forKey:@"max-fps"];

        // 跳帧开关，如果 CPU 解码能力不足，则可以设置成 5，否则会引起音视频不同步，也可以通过设置它来跳帧达到倍速播放
        [options setPlayerOptionIntValue:0 forKey:@"framedrop"];

        // 指定最大宽度
        [options setPlayerOptionIntValue:960 forKey:@"videotoolbox-max-frame-width"];

        // 自动转屏开关
        [options setFormatOptionIntValue:0 forKey:@"auto_convert"];

        // 重连次数
        [options setFormatOptionIntValue:1 forKey:@"reconnect"];

        // 超时时间，timeout 参数只对 http 设置有效，若用 RTMP 设置 timeout，则 IJKPlayer 内部会忽略 timeout 参数。RTMP 的 timeout 参数含义和 http 的不一样
        [options setFormatOptionIntValue:30 * 1000 * 1000 forKey:@"timeout"];

        // 帧速率(fps) （确认非标准帧速率会导致音画不同步，所以只能设定为 15 或者 29.97）
        [options setPlayerOptionIntValue:29.97 forKey:@"r"];
```

- 直播设置。

```
        //如果是 RTSP 协议，则可以优先用 TCP(默认是用 UDP)
        [options setFormatOptionValue:@"tcp" forKey:@"rtsp_transport"];
        //播放前的探测 Size，默认是 1MB，改小一点会出画面更快
        [options setFormatOptionIntValue:1024 * 16 forKey:@"probesize"];
        //播放前的探测时间
        [options setFormatOptionIntValue:50000 forKey:@"analyzeduration"];
        //软解码，更稳定
        [options setPlayerOptionIntValue:0 forKey:@"videotoolbox"];
        //解码参数，画面更清晰
        [options setCodecOptionIntValue:IJK_AVDISCARD_DEFAULT forKey:@"skip_loop_filter"];
        //
        [options setCodecOptionIntValue:IJK_AVDISCARD_DEFAULT forKey:@"skip_frame"];

        Boolean _isLive = true;
        if (_isLive) {
            // Param for living
            [options setPlayerOptionIntValue:3000 forKey:@"max_cached_duration"];   // 最大缓存大小是 3s，可以依据自己的需求修改
            [options setPlayerOptionIntValue:1 forKey:@"infbuf"];  // 无限读
            [options setPlayerOptionIntValue:0 forKey:@"packet-buffering"];  //  关闭播放器缓冲
        } else {
```

```
        // Param for playback
        [options setPlayerOptionIntValue:0 forKey:@"max_cached_duration"];
        [options setPlayerOptionIntValue:0 forKey:@"infbuf"];
        [options setPlayerOptionIntValue:1 forKey:@"packet-buffering"];
    }
```

（5）本地通知。

- 添加本地通知。

```
- (void)installMovieNotificationObservers {
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(loadStateDidChange:)
                                                 name:IJKMPMoviePlayerLoadStateDidCha
ngeNotification
                                               object:_player];
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(moviePlayBackFinish:)
                                                 name:IJKMPMoviePlayerPlaybackDidFini
shNotification
                                               object:_player];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(mediaIsPreparedToPlay
DidChange:)
                                                 name:IJKMPMediaPlaybackIsPreparedToP
layDidChangeNotification
                                               object:_player];

    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(moviePlayBackStateDid
Change:)
                                                 name:IJKMPMoviePlayerPlaybackStateDi
dChangeNotification
                                               object:_player];

}
```

- 移除本地通知。

```
- (void)removeMovieNotificationObservers {
    [[NSNotificationCenter defaultCenter] removeObserver:self
                                                    name:IJKMPMoviePlayerLoadStateDid
ChangeNotification
                                                  object:_player];
    [[NSNotificationCenter defaultCenter] removeObserver:self
                                                    name:IJKMPMoviePlayerPlaybackDidF
inishNotification
```

```
                                                    object:_player];
        [[NSNotificationCenter defaultCenter] removeObserver:self
                                                      name:IJKMPMediaPlaybackIsPrepared
ToPlayDidChangeNotification
                                                    object:_player];
        [[NSNotificationCenter defaultCenter] removeObserver:self
                                                      name:IJKMPMoviePlayerPlaybackStat
eDidChangeNotification
                                                    object:_player];

    }
```

- 通过通知捕获播放状态。

```
    - (void)loadStateDidChange:(NSNotification*)notification {
        IJKMPMovieLoadState loadState = _player.loadState;

        if ((loadState & IJKMPMovieLoadStatePlaythroughOK) != 0) {
            NSLog(@"LoadStateDidChange:
IJKMovieLoadStatePlayThroughOK: %d\n",(int)loadState);
        }else if ((loadState & IJKMPMovieLoadStateStalled) != 0) {
            NSLog(@"loadStateDidChange: IJKMPMovieLoadStateStalled: %d\n",
(int)loadState);
        } else {
            NSLog(@"loadStateDidChange: ???: %d\n", (int)loadState);
        }
    }

    - (void)moviePlayBackFinish:(NSNotification*)notification {
        int reason =[[[notification userInfo]
valueForKey:IJKMPMoviePlayerPlaybackDidFinishReasonUserInfoKey] intValue];
        switch (reason) {
            case IJKMPMovieFinishReasonPlaybackEnded:
                NSLog(@"playbackStateDidChange:
IJKMPMovieFinishReasonPlaybackEnded: %d\n", reason);
                break;

            case IJKMPMovieFinishReasonUserExited:
                NSLog(@"playbackStateDidChange: IJKMPMovieFinishReasonUserExited: %d\n",
reason);
                break;

            case IJKMPMovieFinishReasonPlaybackError:
                NSLog(@"playbackStateDidChange:
IJKMPMovieFinishReasonPlaybackError: %d\n", reason);
                break;
```

```
            default:
                NSLog(@"playbackPlayBackDidFinish: ???: %d\n", reason);
                break;
        }
    }

    - (void)mediaIsPreparedToPlayDidChange:(NSNotification*)notification {
        NSLog(@"mediaIsPrepareToPlayDidChange\n");
    }

    - (void)moviePlayBackStateDidChange:(NSNotification*)notification {
        switch (_player.playbackState) {
            case IJKMPMoviePlaybackStateStopped:
                NSLog(@"IJKMPMoviePlayBackStateDidChange %d: stoped",
(int)_player.playbackState);
                break;

            case IJKMPMoviePlaybackStatePlaying:
                NSLog(@"IJKMPMoviePlayBackStateDidChange %d: playing",
(int)_player.playbackState);
                break;

            case IJKMPMoviePlaybackStatePaused:
                NSLog(@"IJKMPMoviePlayBackStateDidChange %d: paused",
(int)_player.playbackState);
                break;

            case IJKMPMoviePlaybackStateInterrupted:
                NSLog(@"IJKMPMoviePlayBackStateDidChange %d: interrupted",
(int)_player.playbackState);
                break;

            case IJKMPMoviePlaybackStateSeekingForward:
            case IJKMPMoviePlaybackStateSeekingBackward: {
                NSLog(@"IJKMPMoviePlayBackStateDidChange %d: seeking",
(int)_player.playbackState);
                break;
            }

            default: {
                NSLog(@"IJKMPMoviePlayBackStateDidChange %d: unknown",
(int)_player.playbackState);
                break;
            }
        }
    }
```

（6）开始播放。

```
// 启动预播放操作
[self.player prepareToPlay];
// 播放
[self.player play];
```

（7）结束播放。

```
[self.player shutdown];
```

4. 实现效果

实现效果如图 6-13 所示。

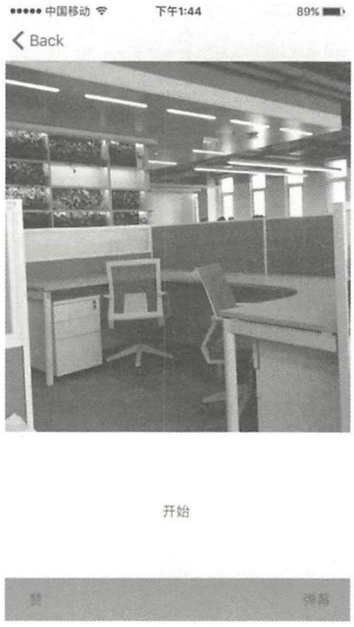


图 6-13

6.6.6 点赞

点赞是观众观看直播时常用的一个功能，实现起来相对简单，可以将其理解为一个自定义的动画效果。

1. 实现步骤

（1）创建显示的 view。

```
        [UIView animateWithDuration:0.5 delay:0.0 usingSpringWithDamping:0.6
initialSpringVelocity:0.8 options:UIViewAnimationOptionCurveEaseOut animations:^{
            self.transform = CGAffineTransformIdentity;
            self.alpha = 0.9;
        } completion:NULL];

        NSInteger i = arc4random_uniform(2);
        NSInteger rotationDirection = 1- (2*i);// -1 OR 1
        NSInteger rotationFraction = arc4random_uniform(10);
        [UIView animateWithDuration:totalAnimationDuration animations:^{
            self.transform = CGAffineTransformMakeRotation(rotationDirection * PI/(16 +
rotationFraction*0.2));
        } completion:NULL];

        UIBezierPath *heartTravelPath = [UIBezierPath bezierPath];
        [heartTravelPath moveToPoint:self.center];
```

（2）创建随机点。

```
        CGPoint endPoint = CGPointMake(heartCenterX + (rotationDirection) *
arc4random_uniform(2*heartSize), viewHeight/6.0 + arc4random_uniform(viewHeight/4.0));

        NSInteger j = arc4random_uniform(2);
        NSInteger travelDirection = 1- (2*j);// -1 OR 1

        CGFloat xDelta = (heartSize/2.0 + arc4random_uniform(2*heartSize)) *
travelDirection;
        CGFloat yDelta = MAX(endPoint.y ,MAX(arc4random_uniform(8*heartSize),
heartSize));
        CGPoint controlPoint1 = CGPointMake(heartCenterX + xDelta, viewHeight - yDelta);
        CGPoint controlPoint2 = CGPointMake(heartCenterX - 2*xDelta, yDelta);

        [heartTravelPath addCurveToPoint:endPoint controlPoint1:controlPoint1
controlPoint2:controlPoint2];
```

（3）创建动画。

```
       CAKeyframeAnimation *keyFrameAnimation = [CAKeyframeAnimation
animationWithKeyPath:@"position"];
        keyFrameAnimation.path = heartTravelPath.CGPath;
        keyFrameAnimation.timingFunction = [CAMediaTimingFunction
functionWithName:kCAMediaTimingFunctionLinear];
        keyFrameAnimation.duration = totalAnimationDuration + endPoint.y/viewHeight;
        [self.layer addAnimation:keyFrameAnimation forKey:@"positionOnPath"];

        //Alpha & remove from superview
        [UIView animateWithDuration:totalAnimationDuration animations:^{
            self.alpha = 0.0;
        } completion:^(BOOL finished) {
```

```
            [self removeFromSuperview];
        }];
```

（4）绘制。

```
        [_strokeColor setStroke];
        [_fillColor setFill];

        CGFloat drawingPadding = 4.0;
        CGFloat curveRadius = floor((CGRectGetWidth(rect) - 2*drawingPadding) / 4.0);
        UIBezierPath *heartPath = [UIBezierPath bezierPath];

        CGPoint tipLocation = CGPointMake(floor(CGRectGetWidth(rect) / 2.0),
CGRectGetHeight(rect) - drawingPadding);
        [heartPath moveToPoint:tipLocation];

        CGPoint topLeftCurveStart = CGPointMake(drawingPadding,
floor(CGRectGetHeight(rect) / 2.4));

        [heartPath addQuadCurveToPoint:topLeftCurveStart
controlPoint:CGPointMake(topLeftCurveStart.x, topLeftCurveStart.y + curveRadius)];

        [heartPath addArcWithCenter:CGPointMake(topLeftCurveStart.x + curveRadius,
topLeftCurveStart.y) radius:curveRadius startAngle:PI endAngle:0 clockwise:YES];

        CGPoint topRightCurveStart = CGPointMake(topLeftCurveStart.x + 2*curveRadius,
topLeftCurveStart.y);
        [heartPath addArcWithCenter:CGPointMake(topRightCurveStart.x + curveRadius,
topRightCurveStart.y) radius:curveRadius startAngle:PI endAngle:0 clockwise:YES];

        CGPoint topRightCurveEnd = CGPointMake(topLeftCurveStart.x + 4*curveRadius,
topRightCurveStart.y);
        [heartPath addQuadCurveToPoint:tipLocation
controlPoint:CGPointMake(topRightCurveEnd.x, topRightCurveEnd.y + curveRadius)];

        [heartPath fill];

        heartPath.lineWidth = 1;
        heartPath.lineCapStyle = kCGLineCapRound;
        heartPath.lineJoinStyle = kCGLineCapRound;
        [heartPath stroke];
```

2. 实现效果

实现效果如图 6-14 所示。

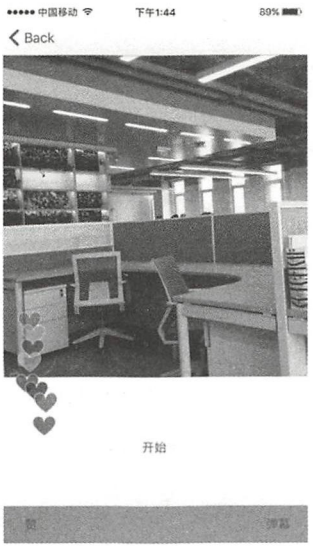


图 6-14

6.6.7　弹幕

弹幕也是直播时观众经常用到的功能，实现起来也并不复杂，也可以借助一些优秀的开源库实现。这里选择了比较好的一个开源框架 BarrageRenderer 来实现弹幕功能。

1. 实现步骤

（1）引用。

```
#import "BarrageRenderer.h"
#import "NSSafeObject.h"
@interface PullFlowViewController ()<BarrageRendererDelegate>{
}
```

（2）初始化弹幕。

```
- (void)initBarrageRenderer
{
    _renderer = [[BarrageRenderer alloc]init];
    _renderer.smoothness = .2f;
    _renderer.delegate = self;
    [_playerView addSubview:_renderer.view];
    _renderer.canvasMargin = UIEdgeInsetsMake(10, 10, 10, 10);
    // 若想为弹幕增加点击功能，则请添加此行代码，并在 Descriptor 中注入行为
    _renderer.view.userInteractionEnabled = YES;
    [_playerView sendSubviewToBack:_renderer.view];
}
```

（3）绘制弹幕样式。

```
    - (BarrageDescriptor
*)walkTextSpriteDescriptorWithDirection:(BarrageWalkDirection)direction
    {
        return [self walkTextSpriteDescriptorWithDirection:direction
side:BarrageWalkSideDefault];
    }

    - (BarrageDescriptor
*)walkTextSpriteDescriptorWithDirection:(BarrageWalkDirection)direction
side:(BarrageWalkSide)side
    {
        BarrageDescriptor * descriptor = [[BarrageDescriptor alloc]init];
        descriptor.spriteName = NSStringFromClass([BarrageWalkTextSprite class]);
        descriptor.params[@"bizMsgId"] = [NSString
stringWithFormat:@"%ld",(long)_index];
        //弹幕显示文字
        descriptor.params[@"text"] = [NSString stringWithFormat:@"过场文字弹
幕:%ld",(long)_index++];
        descriptor.params[@"textColor"] = [UIColor whiteColor];
        descriptor.params[@"speed"] = @(100 * (double)random()/RAND_MAX+50);
        descriptor.params[@"direction"] = @(direction);
        descriptor.params[@"side"] = @(side);
        //单击弹幕提示，可以实现弹幕点赞功能
        descriptor.params[@"clickAction"] = ^(NSDictionary *params){
            NSString *msg = [NSString stringWithFormat:@"弹幕 %@ 被点击
",params[@"bizMsgId"]];
            UIAlertView *alertView = [[UIAlertView alloc]initWithTitle:@"提示"
message:msg delegate:nil cancelButtonTitle:@"取消" otherButtonTitles:nil];
            [alertView show];
        };
        return descriptor;
    }
```

（4）弹幕显示方式。

```
     [_renderer receive:[self
walkTextSpriteDescriptorWithDirection:BarrageWalkDirectionR2L side:BarrageWalkSideLeft]];
    [_renderer receive:[self walkTextSpriteDescriptorWithDirection:BarrageWalkDirectionR2L
side:BarrageWalkSideDefault]];
```

（5）实现回调方法。

```
    - (void)barrageRenderer:(BarrageRenderer *)renderer
spriteStage:(BarrageSpriteStage)stage spriteParams:(NSDictionary *)params
    {
        NSString *subid = [params[@"identifier"] substringToIndex:8];
        if (stage == BarrageSpriteStageBegin) {
```

```
            NSLog(@"id:%@,bizMsgId:%@ =>进入",subid,params[@"bizMsgId"]);
        } else if (stage == BarrageSpriteStageEnd) {
            NSLog(@"id:%@,bizMsgId:%@ =>离开",subid,params[@"bizMsgId"]);
            /* 注释代码演示了如何复制一条弹幕
             BarrageDescriptor * descriptor = [[BarrageDescriptor alloc]init];
             descriptor.spriteName = NSStringFromClass([BarrageWalkTextSprite class]);
             [descriptor.params addEntriesFromDictionary:params];
             descriptor.params[@"delay"] = @(0);
             [renderer receive:descriptor];
             */
        }
    }
```

（6）开始弹幕。

```
    [_renderer start];
     [_timer invalidate];
     NSSafeObject * safeObj = [[NSSafeObject alloc]initWithObject:self
withSelector:@selector(autoSendBarrage)];
     _timer = [NSTimer scheduledTimerWithTimeInterval:0.5 target:safeObj
selector:@selector(excute) userInfo:nil repeats:YES];
```

（7）结束弹幕。

```
    [_rendererstop];
```

2. 实现效果

实现效果如图 6-15 所示。

图 6-15

6.7 本章小结

本章介绍了在 iOS 端实现视频直播的整个过程和实现方式。在实现过程中用到了 3 个开源框架：GPUImage、LFLiveKit 和 IJKPlayer，这些框架对初学者和开发者有很大的帮助。

最后，本章介绍了和直播相关的一些内容。

- 采集数据：在 iOS 平台上采集音/视频数据，需要使用 AVFoundation.framework 框架，从 CaptureSession 会话的回调中获取音/视频数据。
- 传输层协议：主要采用 RTMP 协议（默认端口为 1935，采用 TCP 协议）和 HLS 协议。
- 音/视频编码、解码：使用 FFmpege 编码、解码。
- 视频编码格式：H.265、H.264、MPEG-4 等，封装容器有 TS、MKV、AVI、MP4 等。
- 音频编码格式：G.711、AAC、Opus 等，封装容器有 MP3、OGG、AAC 等。
- 渲染工具：采用 OpenGL 渲染 YUV 数据，呈现视频画面。将 PCM 送入设备的硬件资源播放，产生声音。iOS 播放流式音频，使用 AudioQueue 的方式，即利用 AudioToolbox.framework 框架。

第 7 章

Web 端解决方案

视频直播是近两年互联网行业中很火的一个板块，而 RTMP 是目前市面上实现视频直播所采用的比较主流的数据传输方式。

一般来说，视频主播通过 OBS 等推流软件，将摄像头捕捉的视频通过 RTMP 协议传输到指定的服务器地址，服务器将接收到的视频流以 m3u8 格式保存，客户端再通过拉取 RTMP 视频流的方式获取视频数据并播放。

以上就是一个视频直播的基本模型。如果想直接在浏览器中向 RTMP 服务器推流，该如何实现呢？

本章主要介绍在浏览器中向服务器推送 RTMP 视频流的实现方式。

7.1 Adobe Flash Player

Flash 是由 Macromedia 公司推出的一种交互式矢量图和 Web 动画的标准，之后被 Adobe 公司所收购，如图 7-1 所示。网页设计者使用 Flash 可以创作出既漂亮又可以改变尺寸的导航界面及其他奇特的效果。

图 7-1

Flash 的前身是 Future Wave 的 Future Splash，它是世界上第一款用于设计和编辑 Flash 文档的商用 2D 矢量动画软件。1996 年 11 月，Macromedia 收购了 Future Wave，并将其更名为 Flash。后来 Flash 于 2005 年被 Adobe 公司所收购。2012 年 8 月 15 日，Flash 退出 Android 平台。2015 年 12 月 1 日，Adobe 升级了 Flash CC 2015 动画软件，并更名为 Animate CC 2015.5，使其与 Flash 技术保持一致。

7.1.1 Flash Player

Flash Player 是一个多媒体动画播放器，用于播放小型、快速、交互式的动画，以及动态标志和 Macromedia Flash 制作的图像。这个播放器体积很小，下载很快。Flash Player 还支持高质量的 MP3 音频流、文本输入字段、交互式界面等。使用 Flash Player 最新版本可以观看所有的 Flash 格式。网页上的多媒体内容几乎都是 Flash 格式的。

2003 年 8 月 25 日，Flash 的开发者 Macromedia 推出了 Flash MX 2004。新的 Flash MX 2004 增加了对移动设备和手机的支持，这使得 Flash 可以直接运行于手机上。随着手机性能的大幅提升，目前大部分智能手机完全能够支持 Flash。同时，Flash Player 具有 Java 这样的跨平台功能，所以无论使用何种平台，只要安装了 Flash Player，就能保证最终的显示效果是一致的。但是目前，随着技术的进步，Flash 已经渐渐退出了移动端，移动端的动画播放采用了更新、更轻巧的技术。但 Flash 作为 PC 端向下兼容的解决方案，是必不可少的。

7.1.2 为什么要使用 Flash

一提到 Web 端推送 RTMP 协议视频流，就不得不提 Flash 了。

RTMP 实时消息传送协议是 Adobe Systems 公司为 Flash 的播放器和服务器之间的音频、视频和数据传输开发的开放协议，如图 7-2 所示。

RTMP 有多种变种：

- RTMP 工作在 TCP 之上，默认使用端口 1935。
- RTMPE 在 RTMP 的基础上增加了加密功能。
- RTMPT 被封装在 HTTP 请求之上，可穿透防火墙。
- RTMPS 类似 RTMPT，增加了 TLS/SSL 的安全功能。

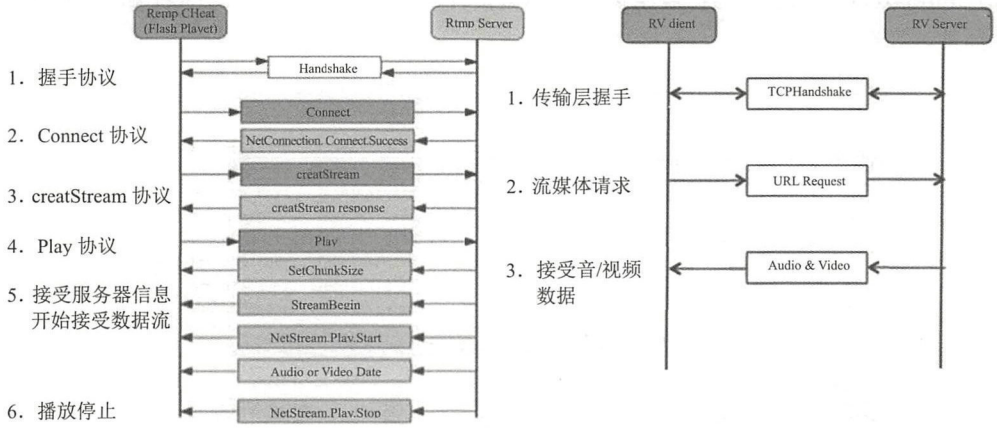


图 7-2

目前，市面上的大部分浏览器都可以很好地支持 Flash，使用 Flash 可以避免很多使用新技术带来的兼容问题。而且，RTMP 正是为了 Flash 而生的！只需要几行简单的 JavaScript 代码，浏览器就可以轻松支持 Flash。

7.2　ActionScript 与 Flex

Flex 是 Adobe 提供的一款 ActionScript 开发框架，其提供了丰富的 API，并能将 ActionScript 编译成可被 Flash Player 执行的 SWF 文件。在开始工作之前，我们需要在开发环境中下载并安装 Flex SDK。

7.2.1　Flex 环境的搭建

Flex 环境的搭建包括以下几步（建议读者使用与本书相同的软件版本）：

- 下载并安装 JDK（本书使用的版本为 Sun JDK 6）。
- 下载并解压 Eclipse（本书使用的版本为 Eclipse Ganymede J2EE，含 WTP 插件）。
- 下载并安装 Flex Builder Eclipse 插件版（本书使用的版本为 Flex Builder 3.0.1）。
- 下载并安装 Tomcat（本书使用的版本为 Tomcat 6.0.18）。
- 下载并安装 FireFox（考虑到 Flex 3.0 和一些插件的兼容性问题，本书使用的版本为 FireFox 2.0.0.17）。

Flex 代码是在运行 Flash Player 的 SWF 文件之后编译的，查看一些 SWF 文件在运行时输出的调试信息比较困难。所以，在安装基本软件之后，必须安装一个 FireFox 插件来调试 Flex。

打开 FireFox，在 FireFox 的扩展组件站点中搜索并安装 HttpFox、Flash Tracer 和 Cache Status 这 3 个插件，如图 7-3 所示。

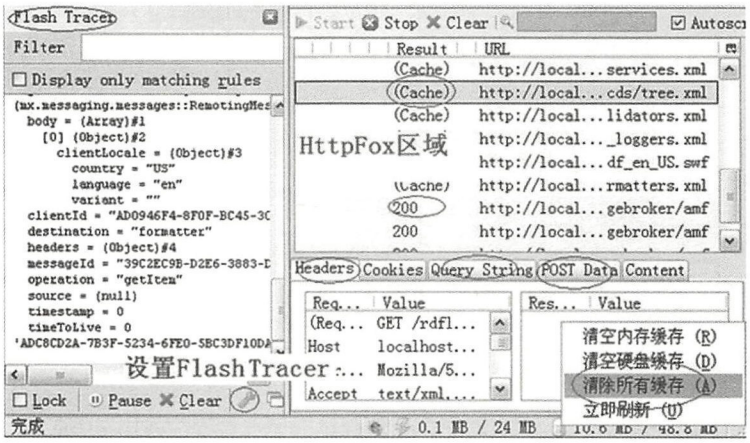


图 7-3

安装 Flash Player 的 Debug 版本后，Flash Tracer 可以显示在程序中使用 trace()语句输出的调试信息。使用 HttpFox 插件不仅可以查看 HTTP 通信的过程和数据，还能查看来自缓存的内容。另外，缓存状态插件也方便我们轻松地管理缓存。在 Flex 开发过程中，经常需要清除缓存的内容，这样可以快速看到最新的效果。

接下来，打开 Flex Builder，选择“Window→Preferences→Server→Runtime Environments”选项并设置 Tomcat，选择“Window→Preferences→General→Web Browser”选项，将浏览器设置为外部浏览器 FireFox，如图 7-4 和图 7-5 所示。

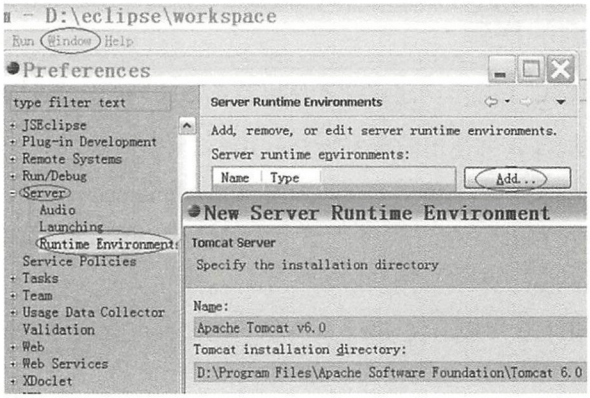


图 7-4

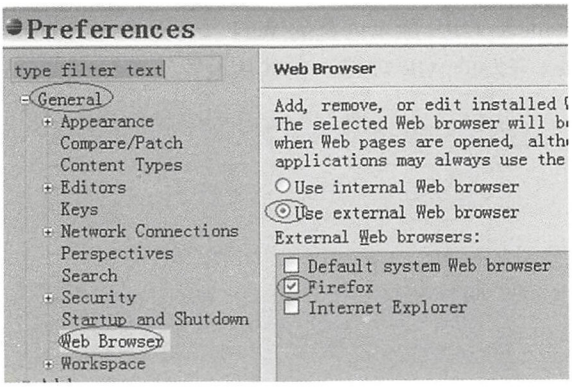


图 7-5

到此为止，开发环境就搭建完毕。

7.2.2　Flex 项目的创建

1. 创建 Flex 项目

打开 Flex Builder，新建一个 Flex Project，如图 7-6 所示。

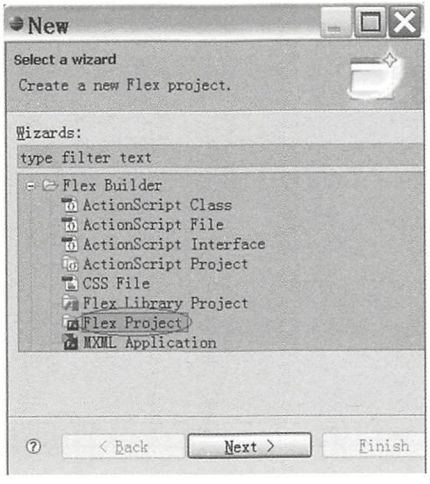


图 7-6

再设置页面，如图 7-7 所示，选择项目类型为 Web application，关于 AIR 应用的基本知识，可以参考 IBM developerWorks 中的一篇文章:《使用 Adobe AIR 和 Dojo 开发基于 Ajax 的 Mashup 应用》。对于 Application server type，这里以 J2EE 为例，不使用 Use remote object access service。

最后，使用 Eclipse Ganymede J2EE 版本内置的 WTP（Web Tools Platform）创建一个后端使用 Java 开发，前端使用 Flex 开发的 AIR 项目。在默认设置下，src 是 Java 代码的源代码文件夹。

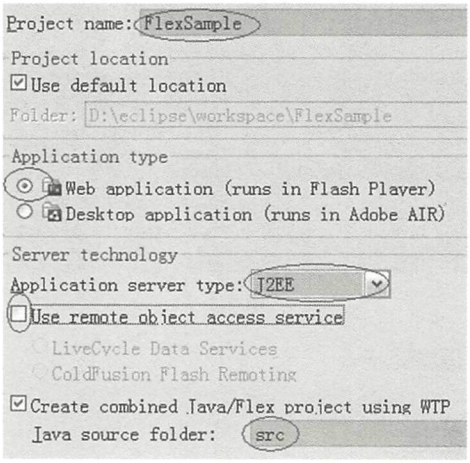


图 7-7

在随后的设置页面中，配置项目运行时的 J2EE Server 为我们在安装配置开发环境中配置的 Apache Tomcat v6.0，如图 7-8 所示。

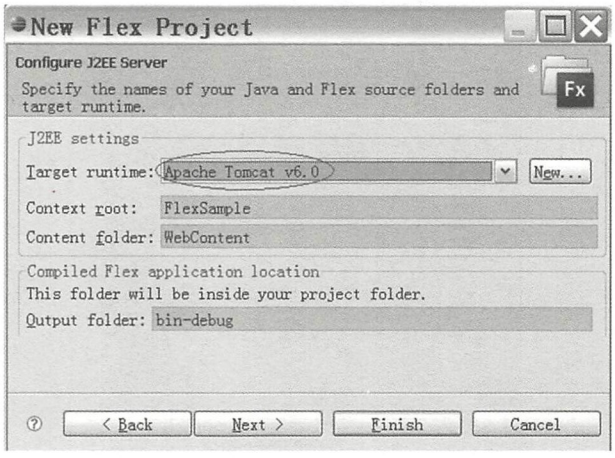


图 7-8

单击“Next”按钮，之后打开的对话框中的所有内容都是默认设置的。如图 7-9 所示，Main source folder 是设置默认的 Flex 代码（包括 MXML 和 ActionScript）的源文件夹。Main application file 是项目默认的主应用文件。Output folder URL 是项目运行在我们配置的 Tomcat 上的 URL，如图 7-9 所示。

完成后，让我们看一看这个项目的组成部分（见图 7-10）：flex_src 是默认的 Flex 源位置，flex_libs 是 Flex 存储的其他第三方包的默认路径。类似于 Web 应用程序的 lib 文件夹。src 是 Java 代码的位置。WebContent 文件夹的结构与由 WTP 创建的 Web 项目的结构完全相同。在默认输出路径 bin-debug 文件夹中，可以看到 Flex Builder 自动生成的 FlexSample.mxml 文件已被自动编译为 FlexSample.swf 文件，如图 7-10 所示。

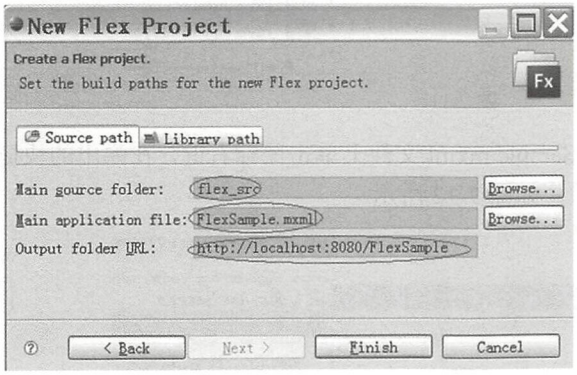


图 7-9

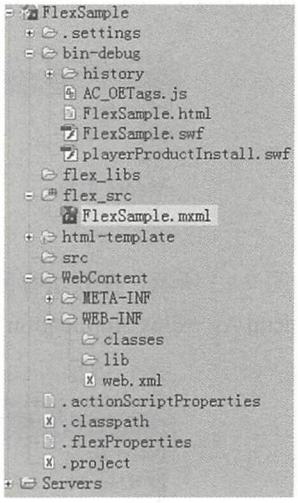


图 7-10

接下来，向新创建的项目中添加一些内容并让它运行：双击 FlexSample.mxml 文件，在其中添加一个最基本的 Flex 组件——Label，并且在该应用初始化的时候调用 init()方法。我们在 init()中用 trace()语句输出调试信息。

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
     initialize="init()">
     <mx:Script>
         <![CDATA[
             private function init():void
             {
                 var i:int = 0;
                 i++;
                 trace("i="+i);
             }
         ]]>
     </mx:Script>
    <mx:Label text="Hello World!" />
```

```
</mx:Application>
```

2. 运行、部署以及调试

在要运行的项目文件上单击鼠标右键，在弹出的快捷菜单中选择“Run As→Run On Server”选项，如图 7-11 所示。在打开的对话框中保持选择默认选项，之后可以看到该项目被部署到配置的 tomcat 6 中，并且 Flex Builder 会自动打开一个之前配置的外部 FireFox 窗口。

图 7-11

如图 7-12 所示，在要运行的 FlexSample.mxml 文件上单击鼠标右键，在弹出的快捷菜单中选择“Run As→Flex Application”选项，如图 7-12 所示。

图 7-12

如果没有意外，则应该看到以下内容：Flash Tracer 插件使用 trace()语句输出调试信息——“Hello World!”，并且显示在屏幕上。另外，打开 FireFox 的 HttpFox 插件，也可以看到 FlexSample.swf 是在运行时加载的，如图 7-13 所示。

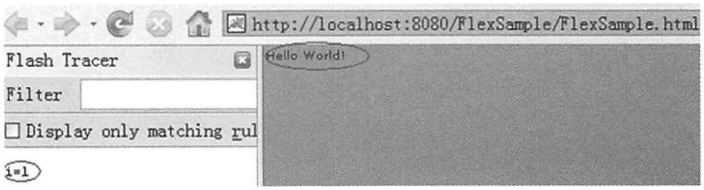


图 7-13

如果想在运行项目时观察变量的值，就像调试 Java 代码一样，该怎么做呢？首先，在 ActionScript 代码中添加断点，就像我们在 Java 代码中设置断点一样，如图 7-14 所示。

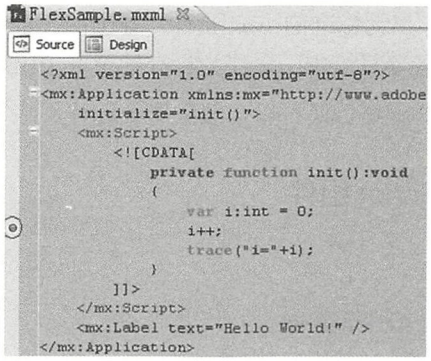


图 7-14

然后选择 FlexSample.mxml 文件，单击鼠标右键，在弹出的快捷菜单中选择“Debug As→Flex Application”选项，调试运行，如图 7-15 所示。

图 7-15

当系统弹出提示信息时，切换到 Flex 的调试视图，如图 7-16 所示。然后我们就可以轻松地调试 Flex 代码了，就像调试 Java 代码，如图 7-16 所示。

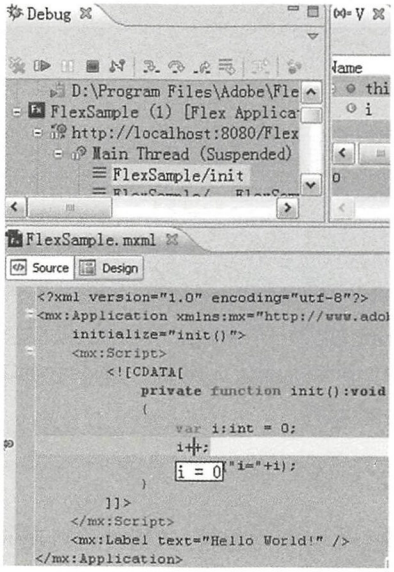


图 7-16

使用 WTP 可以将项目导出为标准的 war 文件。导出的 war 文件可以被放置在 Tomcat 的 webapps 目录或其他 Web 服务器目录中，以作为标准的 Web 应用程序进行部署。

到目前为止，我们编译的程序已经可以在服务器上运行了。

7.2.3 使用 ActionScript 组件

1. Flex 的核心

Flex 的核心是 MXML 和 ActionScript。

MXML 用于为属于表示层的 Flex 应用程序制作 UI 布局，通过编辑到 ActionScript 中并生成 ActionScript 类文件，最终在 Flash Player 上运行。

所以，ActionScript 仍然是 Flex 的核心。在 Flex 中，ActionScript 是一个包含组件（容器和控件）、管理器类、数据服务类和所有其他函数的类的类库。

2. ActionScript 常用的三种方式

（1）内联方式。

```
    <mx:Button label="Say Hello" click="mx.controls.Alert.show('Hello,Flying')"/>
```

（2）级联方式。

```
    <mx:Button label="Say Hello" click="sayHello('Flying')"/>
    <mx:Script>
    <![CDATA[ import mx.controls.Alert; private function
sayHello(param_name:String):void { Alert.show("Hello, "+param_name); } ]]></mx:Script>
```

（3）外联方式。

```
     <mx:Script source="myFunction.as"/>

    <mx:Button label="Say Hello" click="sayHello('Flying');"/>

     // myFunction.asimport mx.controls.Alert; private function
sayHello(param_name:String):void { mx.controls.Alert.show("Hello, "+param_name); }
```

上面的方法是用 ActionScript 创建一个新的、单独的 ascf 文件，然后为 Script 元素的源属性值调用元素设置方法，并且可以在方法中传入参数。这个文件可以在多个文件中调用，以实现 ActionScript 方法在多个文件中的复用。

3. ActionScript 构建组件

可以使用 ActionScript 创建一个可复用组件，该组件可以在 Flex 应用程序中作为标签引用。在 ActionScript 中创建的组件可以包含图像元素、自定义的业务逻辑，甚至可以扩展现有的 Flex

组件。在 ActionScript 中，Flex 组件实现了类层次结构，每个组件都是 Action 类的一个实例。

所有 Flex 可视组件都是从 UIComponent 类派生出来的。要创建自己的组件，则可以创建一个继承 UIComponent 或 UIComponent 子类的类。是否将类用作自定义组件的超类取决于你想要实现的内容。例如，你可能需要一个自定义按钮控件，那么你可以创建 UIComponent 类的子类，然后覆盖 Flex Button 类的所有功能。比较好的创建自定义按钮控件的方法是创建 Flex 按钮组件的子类，并在自定义类中对其进行修改。下面是具体的代码（注：一切从可重用性方面考虑，否则不需要构建组件），仅供参考。

（1）PaddedPanel.as。

```
<SPAN style="FONT-SIZE: large">package components
{
    import mx.containers.Panel;

    public class PaddedPanel extends Panel
    {

        public function PaddedPanel()
        {
            // 调用超类的构造函数
            super();

            // Give the panel a uniform 10-pixel
            // padding on all four sides.
            setStyle("paddingLeft", 10);
            setStyle("paddingRight", 10);
            setStyle("paddingTop", 10);
            setStyle("paddingBottom", 10);
        }

    }
}</SPAN>
```

（2）NumericDisplay.as。

```
<SPAN style="FONT-SIZE: large">package components
{
    import flash.events.Event;
    import flash.events.MouseEvent;

    import mx.containers.Tile;
    import mx.containers.VBox;
    import mx.controls.Button;
    import mx.controls.TextInput;
    import mx.events.FlexEvent;
    public class NumericDisplay extends VBox
```

```
{
    private var display:TextInput;
    private var buttonsTile:Tile;
    // 将_numButtons 属性公开给 Flex Builder 3 中的 View
    [Inspectable(defaultValue=10)]
    private var _numButtons:uint = 10;
   private var _max:int=90;

   public function set max(v:int):void{

    }


    public function get max():int{

     return _max;
    }



    public function NumericDisplay()
    {
        addEventListener(FlexEvent.INITIALIZE, initializeHandler);
    }

    // numButtons 是一个公共属性，用于显示确定的按钮数
    [Bindable(event="numButtonsChange")]
    public function get numButtons():uint
    {
        return _numButtons;
    }
   //当 numButtons 被赋值时，触发 numButtonsChanage 事件，通知所有被绑定的 getter 方法执行
一次
    public function set numButtons(value:uint):void
    {
        _numButtons = value;
        dispatchEvent(new Event("numButtonsChange"));
    }

    // 初始化组件时调用
    private function initializeHandler(event:FlexEvent):void
    {
        // 显示组件
        paint();
    }
    // 将单击按钮的标签添加到显示中
    private function buttonClickHandler(event:MouseEvent):void
```

```
        {
            display.text += (event.target as Button).label;
        }

        // 显示组件
        private function paint():void
        {
            // 创建显示次数
            display = new TextInput();
            display.width=185;
            addChild(display);
            // 创建一个 Tile 容器来容纳按钮
           buttonsTile = new Tile();
            addChild (buttonsTile);

            // 创建按钮并将其添加到 Tile 容器
           for (var i:uint = 0; i < _numButtons; i++)
            {
                var currentButton:Button = new Button();
                currentButton.label = i.toString();
                currentButton.addEventListener(MouseEvent.CLICK,
buttonClickHandler);
                buttonsTile.addChild (currentButton);
            }
        }
    }
}</SPAN>
```

（3）CountryComboBox.as。

```
<span style="font-size: large;">package components
{
    import mx.controls.ComboBox;
    import flash.events.Event;

    public class CountryComboBox extends ComboBox
    {

        private var countryArrayShort:Array = ["US", "UK"];
        private var countryArrayLong:Array = ["United States", "United Kingdom"];

        // 确定显示状态
        [Inspectable(defaultValue=true)]
        private var _useShortNames:Boolean = true;

        public function set useShortNames(state:Boolean):void
        {
```

```
            // 调用方法根据名称长度设置数据

            if (state)
            {
                this.dataProvider = countryArrayShort;
            }

            else
            {
                this.dataProvider = countryArrayLong;
            }

            // 值更改时调度事件（用于数据绑定）

            dispatchEvent(new Event("changeUseShortNames"));
        }

        // 允许其他组件绑定到此属性
        [Bindable(event="changeUseShortNames")]

        public function get useShortNames():Boolean
        {
            return _useShortNames;
        }
    }

}</span>
```

（4）主应用程序 MXML。

```
    <?xml version="1.0" encoding="utf-8"?>
    <SPAN style="FONT-SIZE: large"><s:Application
xmlns:fx="http://ns.adobe.com/mxml/2009"
                xmlns:s="library://ns.adobe.com/flex/spark"
                xmlns:mx="library://ns.adobe.com/flex/mx"
                minWidth="955"
                minHeight="600"
                xmlns:components="components.*">
    <s:layout>
        <s:BasicLayout/>
    </s:layout>
    <fx:Script>
        <![CDATA[
            import flash.events.Event;

            private function handleCloseEvent(eventObj:Event):void
            {
```

```
                status.text = "You selected: \r" + countries.selectedItem as String;
            }
        ]]>
    </fx:Script>
    <fx:Declarations>
        <!-- 将非可视元素（例如服务、值对象）放在此处 -->
    </fx:Declarations>
    <components:PaddedPanel
        title="Custom component inheritance"
        >
        <components:CountryComboBox
            id="countries"
            useShortNames="false"

            />

        <!--
        使用数据绑定显示属性的最新状态
        -->
        <mx:Label text="useShortNames = {countries.useShortNames}"/>

        <mx:ControlBar horizontalAlign="right">

            <mx:Button
                label="Toggle Display"
                click="countries.useShortNames = !countries.useShortNames;"
                />
        <mx:Text id="status" text="Please select a country from the list above."
width="136"/>
        <components:NumericDisplay numButtons="10"/>
    </mx:ControlBar>
    </components:PaddedPanel>

</s:Application>
</SPAN>
```

4. 灵活创建自定义组件

使用 ActionScript 组件的一般目的是创建可配置和可重用的组件。例如，创建一个具有属性、可分配时间、可定义新样式，以及具有自定义 ActionScript 组件等。

在创建自定义 ActionScript 组件时，其中的注意事项之一是可重用性，即是否要创建一个紧密耦合的组件，它是与某个应用程序紧密耦合的组件，还是可以在多个应用程序中重用的组件。

编写与某个应用程序紧密耦合的组件，通常依赖于应用程序的结构、变量名称或其他细节的组件。如果要更改此应用程序，则可能需要修改与之紧密耦合的组件以反映此更改。

5．松散耦合

要设计一个松散耦合的组件以供重用，组件需要有明确可辨识的接口，指定如何将信息传递给组件，以及如何将结果传递回应用程序。

典型的松散耦合的组件使用属性将信息传递给组件。这些属性由默认访问器（setter 和 getter 方法）定义，并指定参数的类型。例如 CountryComboBox 自定义组件定义了一个公共的 _userShortNames 属性，该属性通过使用 get userShortNames()和 set useShortNames()访问器方法来公开私有属性_userShortNames。

私有属性_userShortNames的Inspectable元数据标记定义了属性，该属性被显示在AdobeFlex Builder 的属性提示中，并被标记在内在函数中。也可以使用此元数据标记来限制允许的属性值。

注意：所有公共属性都会出现在 MXML 代码提示和属性检查器中。如果有关于可以辅助阅读的代码提示或属性检查器中属性的附加信息（如枚举值，或者一些文件路径的字符串），那么附加信息也会被添加到 Inspectable 元素的 Data 中。

6．利用代码提示和属性检查器

MXML 代码提示和属性检查器来自相同的数据，所以，如果显示其中一个，那么另一个应该也是显示的状态。

另外，如果 ActionScript 代码暗示元数据不起作用，则可以随时在 ActionScript 中看到相应的代码提示，具体内容取决于当前的上下文。Flex Builder 使用 ActionScript 代码提示。

定义组件以将信息返回给主应用程序的最佳办法是设计包含要返回的数据的组件分发事件。这样，主函数可以定义事件监听器来处理事件并采取适当的行动。也可以在事件中使用数据绑定，比如，绑定属性使用 Bindable 元数据标签 userShortName 进行编程。隐式的 userShortName 属性设置器发送更改事件，在此过程中使用 Flex 框架的内部机制来使数据绑定正常工作。

7.2.4 NetStream 对象

NetStream 对象是视频流实现相关操作的关键部分，当然，同样重要的还有 NetConnection 类。

NetStream 对象在 Flash Player 或 AIR 应用程序与 Flash Media Server 之间或者 Flash Player 或 AIR 应用程序与本地文件系统之间打开单向流式连接。NetStream 对象是 NetConnection 对象中的通道。此通道可以使用 NetStream.publish()方法发布流或订阅发布的流，并使用

NetStream.play()方法接收数据，可以发布或播放实时数据及以前记录的数据，还可以使用 NetStream 对象向所有订阅的客户端发送文本消息（请参阅 NetStream.send()方法）。

播放外部视频文件比在 SWF 文件中嵌入视频具有许多优点，例如，具有更好的性能和内存管理，以及独立的视频和 SWF 帧频。

NetStream 对象提供的方法和属性可用于跟踪、加载和播放文件，并允许用户控制播放文件（停止或暂停等）。

以下是发布实时音/视频的工作流程。

- 创建一个 NetConnection 对象。
- 使用 NetConnection.connect()方法连接到服务器中的应用程序实例。
- 创建一个 NetStream 对象，以便在连接中创建数据流。
- 使用 NetStream.attachAudio()方法通过流捕获并发送音频，然后使用 NetStream.attachCamera()方法捕获并发送视频。
- 使用 NetStream.publish()方法为该流提供唯一的名称，然后使用该流将数据发送到服务器中，以便其他用户可以接收数据。还可以记录发布数据的时间，以便用户稍后可以播放。

订阅此流的文件将在调用 play()方法时使用传递给 publish()方法的名称，并与发布者调用相同的 NetConnection.connect()方法。它们必须调用 Video.attachNetStream()方法来传输视频，然后调用 NetStream.play()方法来播放视频。

使用 Flash Media Server 的关键帧创建 NetConnection 和 NetStream 对象后，可以使用 NetStream.send()方法将现场音/视频流的元数据添加到服务器中。元数据可以是诸如视频的高度或宽度、持续时间、创建者的名字等信息。要定义元数据，需要使用特殊处理程序名称 @setDataFrame 作为 NetStream.send()方法的第一个参数。接收 Live Streaming 的客户端还需要定义 onMetaData 事件处理程序以从流中检索元数据。

7.2.5　获取视频流

NetStream 对象通过 NetConnection 对象打开一个单向流量通道。使用 NetStream 对象可以执行以下操作。

- 调用 NetStream.play()方法，从本地磁盘、Web 服务器或 Flash Media Server 中播放媒体文件。
- 调用 NetStream.publish()方法，将视/音频和数据流发布到 Flash Media Server 中。
- 调用 NetStream.send()方法，将数据消息发送到所有订阅客户端中。

- 调用 NetStream.send()方法，向实时流添加元数据。
- 调用 NetStream.appendBytes()方法，将 ByteArray 数据传递给 NetStream。

注意：不能通过同一个 NetStream 对象播放和发布流。

直接使用 ActionScript，可以通过摄像头、麦克风和流媒体数据捕获网络中的素材和声音到 RTMP 服务器中。

```
cam = Camera.getCamera();
 ns = new NetStream(nc);
 ns.attachCamera(cam);
 ns.publish(name, "live");
```

Adobe AIR 和 Flash Player 9.0.115.0 及其更高版本，支持从标准 MPEG-4 容器格式派生出来的文件格式。如果包含 H.264 视频和/或 HE-AAC V2 编码音频，则这些文件包括 F4V、MP4、M4A、MOV、MP4V、3GP 和 3G2 格式。在相同编码配置文件下，H.264 可以比 Sorenson 或 On2 以更低的比特率提供更高质量的视频。AAC 是 MPEG-4 视频标准中定义的标准音频格式。HE-AAC V2 是 AAC 的扩展，使用带通复制（SBR）和参数立体（PS）技术来提高低码率下的编码效率。

有关支持的编解码器和文件格式的信息，请参阅：

- 《Flash Media Server 帮助文档》。
- 《探索 Flash Player 对高清 H.264 视频和 AAC 音频的支持》。
- 《FLV/F4V 开放规范文档》。

Flash Media Server、F4V 文件和 FLV 文件可以在流式传输或播放期间发送包含特定数据点数据的事件对象。在播放过程中，可以使用两种方法处理来自数据流或 FLV 文件的数据：

将客户端属性与事件处理程序关联以接收数据对象。

使用 NetStream.client 属性分配对象可以调用特定的数据处理函数。分配给 NetStream.client 属性的对象可以侦听以下数据点：

- onCuePoint()
- onImageData()
- onMetaData()
- onPlayStatus()
- onSeekPoint()

- onTextData()
- onXMPData()

在编写这些函数的过程中，处理了播放过程中从流中返回的数据对象。将客户端属性与 NetStream 类的子类相关联并写入事件处理程序中，可以接收数据对象。NetStream 是一个密封的类，也就是说，不能在运行程序时向 NetStream 类添加属性或方法。但是，可以创建 NetStream 的子类并定义事件处理程序，也可以使这个子类动态并将事件处理程序添加到子类的实例中。在使用对象复制、直接路由或发布 API 之前，应等待接收到的 NetGroup.Neighbor.Connect 事件。

注意：要通过音频文件（如 MP3 文件）发送数据，应使用 Sound 类将音频文件与 Sound 对象相关联，然后使用 Sound.id3 属性读取声音文件中的元数据。

7.2.6　实例：使用 as 实现一个基础的推流器

```
package {
import flash.display.MovieClip;
import flash.external.ExternalInterface;
import flash.net.NetConnection;
import flash.events.NetStatusEvent;
import flash.net.NetStream;
import flash.media.Video;
import flash.media.Camera;
import flash.media.Microphone;

//import flash.media.H264Profile;
//import flash.media.H264VideoStreamSettings;

public class RTMPStreamer extends MovieClip {

    internal var nc:NetConnection;
    internal var ns:NetStream;
    internal var nsPlayer:NetStream;
    internal var vidPlayer:Video;
    internal var cam:Camera;
    internal var mic:Microphone;

    internal var _camWidth:int = 640;
    internal var _camHeight:int = 480;
    internal var _camFps:int = 15;
    internal var _camFrameInterval:int = 25;
```

```
internal var _camBandwidth:int = 200000;
internal var _camQuality:int = 90;

internal var _micQuality:int = 9;
internal var _micRate:int = 44;

internal var _screenWidth:int = 320;
internal var _screenHeight:int = 240;
internal var _screenX:int = 0;
internal var _screenY:int = 0;


public function RTMPStreamer() {
    ExternalInterface.addCallback("setScreenSize", setScreenSize);
    ExternalInterface.addCallback("setScreenPosition", setScreenPosition);
    ExternalInterface.addCallback("setCamMode", setCamMode);
    ExternalInterface.addCallback("setCamFrameInterval", setCamFrameInterval);
    ExternalInterface.addCallback("setCamQuality", setCamQuality);

    ExternalInterface.addCallback("setMicQuality", setMicQuality);
    ExternalInterface.addCallback("setMicRate", setMicRate);

    ExternalInterface.addCallback("publish", publish);
    ExternalInterface.addCallback("play", playVideo);
    ExternalInterface.addCallback("disconnect", disconnect);

    ExternalInterface.call("setSWFIsReady");
}

public function setScreenSize(width:int, height:int):void {
    _screenWidth = width;
    _screenHeight = height;
}

public function setScreenPosition(x:int, y:int):void {
    _screenX = x;
    _screenY = y;
}

public function setCamMode(width:int, height:int, fps:int):void {
    _camWidth = width;
    _camHeight = height;
    _camFps = fps;
}
```

```
        public function setCamFrameInterval(frameInterval:int):void {
            _camFrameInterval = frameInterval;
        }

        public function setCamQuality(bandwidth:int, quality:int):void {
            _camBandwidth = bandwidth;
            _camQuality = quality;
        }

        public function setMicQuality(quality:int):void {
            _micQuality = quality;
        }

        public function setMicRate(rate:int):void {
            _micRate = rate;
        }

        public function publish(url:String, name:String):void {
            this.connect(url, name, function (name:String):void {
                publishCamera(name);
                displayPublishingVideo();
            });
        }

        public function playVideo(url:String, name:String):void {
            this.connect(url, name, function (name:String):void {
                displayPlaybackVideo(name);
            });
        }

        public function disconnect():void {
            nc.close();
        }

        private function connect(url:String, name:String, callback:Function):void {
            nc = new NetConnection();
            nc.addEventListener(NetStatusEvent.NET_STATUS, function
(event:NetStatusEvent):void {
                ExternalInterface.call("console.log", "try to connect to " + url);
                ExternalInterface.call("console.log", event.info.code);
                if (event.info.code == "NetConnection.Connect.Success") {
                    callback(name);
                }
            });
            nc.connect(url);
```

```
        }

        private function publishCamera(name:String):void {
    //        Cam

            cam = Camera.getCamera();

            /**
             * public function setMode(width:int, height:int, fps:Number,
favorArea:Boolean = true):void
             *  width:int——请求的捕获宽度，以像素为单位，默认值是 160
             *  height:int——请求的捕获高度，以像素为单位，默认值是 120
             *  fps:Number——请求捕获帧速率，每秒帧数，默认值是 15
             */
            cam.setMode(_camWidth, _camHeight, _camFps);

            /**
             * public function setKeyFrameInterval(keyFrameInterval:int):void
             * 传输完整的视频帧的数量（称为帧），而不是被视频压缩算法的插值
             * 默认值是 15，这意味着每 15 帧是关键帧。值为 1，意味着每一帧是关键帧
             * 允许的值为 1~300
             */
            cam.setKeyFrameInterval(_camFrameInterval);

            /**
             * public function setQuality(bandwidth:int, quality:int):void
             * bandwidth:int——指定当前传出视频馈送可以使用的最大带宽量，以字节/秒为单位
             * 为了指定视频，可以使用所需的带宽来保持质量的值
             * 默认值是 16384
             * quality:int——一个整数，指定所需的图片质量级别，由压缩量决定
             * 应用于每个视频帧。可接受的值范围为从 1（最低质量，最大压缩）到 100（最高质量，无压缩）。
指定图片质量可以根据需要而变化，以避免超出带宽
             */
            cam.setQuality(_camBandwidth, _camQuality);

            /**
             * public function setProfileLevel(profile:String, level:String):void
             * 设置视频编码的配置文件和级别
             * 轮廓可能值 h264profile.baseline 和 h264profile.main。默认值是
h264profile.baseline。
             * 其他值将被忽略并导致错误
             * 支持的级别是 1、1b、1.1、1.2、1.3、2、2.1、2.2、3、3.1、3.2、4、4.1、4.2、5 和 5.1
             * 如果分辨率和帧速率有要求，则可能会增加
             */
    //            var h264setting:H264VideoStreamSettings = new
```


```
H264VideoStreamSettings();
    //          h264setting.setProfileLevel(H264Profile.MAIN, 4);


    //          Mic

          mic = Microphone.getMicrophone();

          /*
           * 编码后的语音质量采用 Speex 的时候，可能的值是 0~10。默认值是 6
           * 更高的数字表示更高的质量，但需要更多的带宽，如下表所示
           * 这些比特率值代表净比特率，不包括打包的开销
           * ----------------------------------------
           * Quality value | Required bit rate (Kbps)
           *----------------------------------------
           *      0        |       3.95
           *      1        |       5.75
           *      2        |       7.75
           *      3        |       9.80
           *      4        |       12.8
           *      5        |       16.8
           *      6        |       20.6
           *      7        |       23.8
           *      8        |       27.8
           *      9        |       34.2
           *      10       |       42.2
           *----------------------------------------
           */
          mic.encodeQuality = _micQuality;

          /* 麦克风捕捉声音的速率，在 kHz 级别。可接受的值分别为 5kHz、8kHz、11kHz、22kHz 和 44kHz。
默认值是 8kHz
           * 如果声音捕获设备则支持此值。否则，默认值是 8kHz 以上的下一个可用捕获电平
           * 通常是 11kHz
           *
           */
          mic.rate = _micRate;


          ns = new NetStream(nc);
    //      H.264 Setting
    //      ns.videoStreamSettings = h264setting;
          ns.attachCamera(cam);
          ns.attachAudio(mic);
          ns.publish(name, "live");
```


```
        }

        private function displayPublishingVideo():void {
            vidPlayer = getPlayer();
            vidPlayer.attachCamera(cam);
            addChild(vidPlayer);
        }

        private function displayPlaybackVideo(name:String):void {
            nsPlayer = new NetStream(nc);
            nsPlayer.play(name);
            vidPlayer = getPlayer();
            vidPlayer.attachNetStream(nsPlayer);
            addChild(vidPlayer);
        }

        private function getPlayer():Video {
            vidPlayer = new Video(_screenWidth, _screenHeight);
            vidPlayer.x = _screenX;
            vidPlayer.y = _screenY;

            return vidPlayer;
        }

    }

}
```

7.3 SWFObject

7.3.1 为什么选择 SWFObject

1. SWFObject 的优点

- 比其他嵌入方法更灵活、更高效。
- 无论你是 HTML 开发人员还是 Flash 或 JavaScript 开发人员，都能找到适合自己的方法。
- 打破默认的标签集，以促进用户使用自定义标签。
- 使用性能更高的 JavaScript 方法。
- 使用方便。

2. 为什么 SWFObject 使用 JavaScript

- SWFObject 使用 JavaScript 来克服在页面中使用标签引入 Flash 的问题：它会检测 Flash 播放器的版本，并确定是否显示 Flash 内容或替代为其他内容，以防止以版本过低的 Flash 插件显示 Flash 内容。
- 如果 Flash 插件版本过低，则可以通过操作 dom 显示给用户一些信息。

注意：如果 Flash 插件没有安装，则会以 dom 元素替代 Flash 标签。

- 提供可以快速安装 Adobe 的最新 Flash Player 的通道。
- 提供一套 JavaScript API 来执行常见的 Flash 播放器操作。

3. 该使用静态的还是动态的方法发布

SWFObject 提供了两种不同的方法嵌入 Flash Player：

- 静态发布的方法用标准的标签嵌入 Flash 内容和替代元素，并使用 JavaScript 来解决那些单独用标签无法解决的问题。
- 动态发布方法是基于标签的替代内容，通过 JavaScript 用 Flash 来替换替代内容的方法，前提是当前 Flash 版本和 JavaScript 支持。

（1）静态发布的优点：

- 有利于标准的实际生产。
- 拥有最佳的嵌入性能。
- Flash 内容采用嵌入式机制，不依赖于脚本语言，所以拥有更好的兼容性。
- 如果安装了 Flash 插件，但 JavaScript 已停用，或者浏览器不支持 JavaScript，则仍然可以看到 Flash 内容。
- 可以运行在 Flash 支持困难的设备上，如索尼 PSP。
- RSS 阅读器等自动化工具可以捕捉到 Flash 内容。

（2）动态发布的优点：

- 与脚本应用程序集成良好，可以实现动态 Flash 效果。
- 避免了使用点击激活机制来激活 IE 6/7 和 Opera 9.0 以上浏览器中的活动内容。

7.3.2 静态嵌入 Flash Player

1. 使用 SWFObject 的静态方法嵌入 Flash Player

使用符合标准的代码嵌入 Flash Player 和备用内容。SWFObject 的基本标签使用嵌套对象的方法（用专有 IE 条件注释），以确保跨浏览器支持，并仅针对标签进行优化。

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN""http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
      <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
       <title>SWFObject - step 1`</title>
       <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
      </head>
      <body>
        <div>

          <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" width="780"
height="420">
            <param name="movie" value="myContent.swf" />
            <!--[if !IE]>-->
            <object type="application/x-shockwave-flash" data="myContent.swf"
width="780" height="420">
            <!--<![endif]-->
              <p>Alternative content</p>
            <!--[if !IE]>-->
            </object>
            <!--<![endif]-->
          </object>

        </div>
      </body>
    </html>
```

注意：嵌套对象方法需要一个双重对象定义（外部对象针对 IE 浏览器，其他对象用于其他所有浏览器），需要定义你的对象属性和嵌套的 Param 元素两次。

需要的属性：

- classid（只用于外层元素，值一直是：clsid:D27CDB6E-AE6D-11cf-96B8-444553540000）
- type（只用于内层元素，值一直是：application/x-shockwave-flas）
- data（只用于内层元素，定义 swf 的路径：data="myContent.swf"）
- width（定义 swf 的宽度，内外层都用到）
- height（定义 swf 的高度，内外层都用到）

需要的参数：

- movie(只用于内层元素，定义 swf 的路径：<param name="movie" value= "myContent.swf" />)

建议不要使用 code base 属性指向 Adobe Flash 插件安装程序的 URL 的服务器，因为这是违法的，它限定了只能当前的域来访问。我们建议在替换内容中添加一个提示，以便用户获得更好的体验，而不是下载 Flash。

---

2. 怎么使用 HTML 来配置 Flash 内容

可以在标签中添加下面的属性：

- id
- name
- class
- align

可以用下面的参数：

- play
- loop
- menu
- quality
- scale
- align
- wmode
- bgcolor
- base
- swliveconnect
- flashvars
- devicefont (more info)
- allowscriptaccess (more info here and here)
- seamlesstabbing (more info)
- allowfullscreen (more info)
- allownetworking (more info)

3. 使用替换元素

object 元素允许在其中放置替换元素，这些元素在未安装或不支持 Flash Player 时显示，其内容也将被搜索引擎抓取。总而言之，当你希望你的内容可供没有插件的用户访问时，则应该使用替换内容，这样对搜索引擎友好。如果有访问者的提示，则可以让用户获得更好的体验，而不是直接让用户下载插件。

将 JavaScript 库引在 HTML 页面的头部文件中。SWFObject 库是一个外部 JavaScript 文件，一旦加载完成，SWFObject 就会被执行，并且一旦 dom 元素加载完成，dom 操作就会被执行。所有浏览器都支持（IE、Firefox、Safari 和 Opera 9.0）此操作，此时 onload 事件会被触发。

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN""http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
      <head>
        <title>SWFObject - step 2</title>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />

        <script type="text/javascript" src="swfobject.js"></script>

      </head>
      <body>
        <div>
          <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" width="780"
height="420">
            <param name="movie" value="myContent.swf" />
            <!--[if !IE]>-->
            <object type="application/x-shockwave-flash" data="myContent.swf"
width="780" height="420">
            <!--<![endif]-->
              <p>Alternative content</p>
            <!--[if !IE]>-->
            </object>
            <!--<![endif]-->
          </object>
        </div>
      </body>
    </html>
```

下面用 SWFObject 库注册你的 Flash，并告诉 SWFObject。

具体如何执行呢？可以添加一个唯一的 ID 外部对象标签来定义你的 Flash 或是添加 swfobject.registerObject：

- 第 1 个参数（字符串，必需）：指定标签中使用的 ID。
- 第 2 个参数（字符串，必需）：指定为内容分配的 Flash 版本号。它会激活 Flash 版本

检测，以确定是否显示 Flash 内容或强制通过 dom 操作显示替换内容。Flash 的版本号通常由 4 个部分组成：major.minor.release.build，但 SWFObject 只识别前 3 位，所以“WIN 9,0,18,0”（IE 浏览器）或“Shockwave Flash 9 r18”（其他浏览器）将被翻译为“9.0.18”。如果只想测试主要版本号，则可以省略次要版本号和分发版本号，例如翻译“9”而不是“9.0.0”。

- 第 3 个参数，字符串（可选），可用于启动 Adobe Express 安装程序并指定 Quick Install SWF 文件的 URL。当所需插件版本不可用时，“快速安装”将显示标准的 Flash 插件下载对话框，以替换你的 Flash 内容。项目中默认的 expressInstall.swf 文件被打包在一起，其中还包括相应的 expressInstall.fla 和 AS 文件（位于 src 目录中），可以让你自定义快速安装。

注意：快速安装只会被触发一次（第一次被调用时），它只支持 Windows 和 MAC 平台上的 Flash Player 6.0.65 版本，并且需要的最小尺寸为 310px×137px。

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN""http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
      <head>
        <title>SWFObject v2.0 - step 3</title>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
        <script type="text/javascript" src="swfobject.js"></script>

        <script type="text/javascript">
        swfobject.registerObject("myId", "9.0.0", "expressInstall.swf");
        </script>

      </head>
      <body>
        <div>

          <object id="myId" classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
width="780" height="420">

            <param name="movie" value="myContent.swf" />
            <!--[if !IE]>-->
            <object type="application/x-shockwave-flash" data="myContent.swf"
width="780" height="420">
            <!--<![endif]-->
```

```
            <p>Alternative content</p>
          <!--[if !IE]>-->
          </object>
          <!--<![endif]-->
        </object>
      </div>
    </body>
  </html>
```

- 第 4 个参数（JavaScript 函数，可选）：可用于定义一个回调函数，可以调用该函数来处理插件创建成功或失败时的情况。

```
  <script type="text/javascript">

    swfobject.registerObject("myId", "9.0.115", "expressInstall.swf");
  </script>
```

注意：

1. 可以使用 SWFObject HTML 和 JavaScript 生成器来更快地编写代码。

2. 可以将多个 SWF 文件嵌入一个 HTML 页面中。

3. 引用动态对象元素的最简单方法是使用 JavaScript API：swfobject.getObjectById (objectIdStr)。

7.3.3 动态嵌入 Flash Player

1. 使用 SWFObject 动态嵌入 Flash Player

第 1 步：用符合标准的标签创建要替换内容。SWFObject 动态嵌入遵循逐步增强的原则，当有足够的 JavaScript 和 Flash 插件支持时，用 Flash 代替要替换的内容。下面先定义替代方案，用一个 id 标记它。

```
  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN""http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
    <head>
      <title>SWFObject v2.0 dynamic embed - step 1</title>
      <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    </head>
    <body>
```

```
    <div id="myContent">
      <p>Alternative content</p>
    </div>

  </body>
</html>
```

第 2 步：将 JavaScript 库引在 HTML 页面的头部。SWFObject 库是一个外部的 JavaScript 文件。一旦加载完成，SWFObject 就会被执行，并且一旦 dom 元素加载完成，dom 操作就会被执行。所有浏览器都支持（IE、Firefox、Safari 和 Opera 9.0 以上）dom 操作。onload 事件在被触发后执行。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
  <head>
    <title>SWFObject v2.0 dynamic embed - step 2</title>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />

    <script type="text/javascript" src="swfobject.js"></script>

  </head>
  <body>
    <div id="myContent">
      <p>Alternative content</p>
    </div>
  </body>
</html>
```

第 3 步：使用 JavaScript 嵌入 SWF：

- swfobject.embedSWF（swfUrl，id，width，height，version，expressInstallSwfurl，flashvars，params，attributes，callbackFn）：有 5 个必需的参数和 5 个可选参数。
- swfUrl（string，必需）：指定 SWF 的 URL。
- id（string，必需）：指定包含替换元素的 HTML 元素的 ID，可以用闪光灯的内容替换。
- width（string，必需）：指定 SWF 的宽度。
- height（string，必需）：指定 SWF 的高度。
- version（string，必需）：指定 SWF 发布所需要的 Flash Player 的版本（格式为：major.minor.release 或 major）。
- expressInstallSwfurl（字符串，可选）：指定快速安装路径来激活快速安装。
- flashvars（object，可选）：指定要传递给 Flash 的变量（使用键值对）。
- params（object，可选）：指定嵌入对象的参数（使用键值对）。

- attributes（object，可选）：指定对象的属性（使用键值对）。
- callbackFn（JavaScript 函数，可选）：定义可以调用的回调函数，无论调用 Flash 是成功还是失败。

小提示：要想不破坏参数的顺序，则可以省略可选参数。如果不想使用某个参数，但想使用下一个参数，则可以将其值设置为 false。对于 Flash，其参数和属性也可以使用{} 。

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN""http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
      <head>
        <title>SWFObject v2.0 dynamic embed - step 3</title>
        <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
        <script type="text/javascript" src="swfobject.js"></script>

        <script type="text/javascript">
        swfobject.embedSWF("myContent.swf", "myContent", "300", "120", "9.0.0");
        </script>

      </head>
      <body>
        <div id="myContent">
          <p>Alternative content</p>
        </div>
      </body>
    </html>
```

2. 如何配置 Flash

要配置 Flash，则可以添加以下经常使用的可选属性的对象元素：

- id（如果没有定义，则会自动取替换元素容器的 id）
- align
- name
- styleclass
- class

注意：class 是 ECMA4 中保留的关键字，在 IE 浏览器中会报错，除非用引号把它括起来（如 "class" or 'class'）。出于这个原因，SWFobject 提供了作为一种安全的替代语法类的 styleClass 关键字，如果使用的是 styleClass，则 SWFobject 会自动插入并替换成 class。

```
var attributes = {
   id: "myId",
   align: "left",
   styleclass: "myclass"
};
```

如果你宁愿用 class 代替 styleClass，那么要用引号将其括起来。

```
var attributes = {
   id: "myId",
   align: "left",
   "class": "myclass"
};
```

可以使用下面可选的 Flash 指定的参数：

- play
- loop
- menu
- quality
- scale
- align
- wmode
- bgcolor
- base
- swliveconnect
- flashvars
- devicefont (more info)
- allowscriptaccess (more info here and here)
- seamlesstabbing (more info)
- allowfullscreen (more info)
- allownetworking (more info)

3．使用 JavaScript 对象来定义 flashvars、params 和 object's attributes

最好使用对象表示法来定义对象：

```
<script type="text/javascript">
var flashvars = {};
var params = {};
var attributes = {};

swfobject.embedSWF("myContent.swf", "myContent", "300", "120",
"9.0.0","expressInstall.swf", flashvars, params, attributes);
```

</script>可以用键值对的方式写:

```
var flashvars = {
  name1: "hello",
  name2: "world",
  name3: "foobar"
};
```

也可以用"属性"的形式写:

```
var flashvars = {};
flashvars.name1 = "hello";
flashvars.name2 = "world";
flashvars.name3 = "foobar";
```

还可以直接把参数内容加载至 swfobject.embedSWF()中。

```
<script type="text/javascript">
swfobject.embedSWF("myContent.swf", "myContent", "300", "120",
"9.0.0","expressInstall.swf", {name1:"hello",name2:"world",name3:"foobar"},
{menu:"false"}, {id:"myDynamicContent",name:"myDynamicContent"});
</script>
```

如果你不想使用一个可选的参数，则可以把它定义为 false 或一个空对象。

```
var flashvars = false;var params = {};
```

flashvars 的对象是一个速记符号，为了易用，你可以忽略它，通过 params 对象来指定你的 flashvars。

```
<script type="text/javascript">
var flashvars = false;
var params = {
  menu: "false",
  flashvars: "name1=hello&name2=world&name3=foobar"
};
var attributes = {
  id: "myDynamicContent",
  name: "myDynamicContent"
};
```

```
    swfobject.embedSWF("myContent.swf", "myContent", "300", "120",
"9.0.0","expressInstall.swf", flashvars, params, attributes);
    </script>
```

在从 SWFObject（1.5 版本）迁移到 SWFObject（2.0 版本）时，需要注意以下几点：

- SWFObject（2.0 版本）不能与 SWFObject 1.5 向后兼容。
- 当前首选将所有脚本块插入 HTML 页面的标题中，由于 JavaScript 代码执行的原因，将冲突添加到页面主体（例如是 Flash 而不是替换内容）。
- 库的实际名称是小写的（例如是 swfobjec，而不是 SWFObject）。
- 方法只能通过库访问。

另外，SWFObject 2.0 版本与 JavaScript API 完全不同且更复杂，只要 JavaScript 可用，SWFObject （2.0 版本）就会将整个替换标记的内容（包括所引用的 HTML 容器元素）替换为 Flash 版本支持的 Flash 内容。但是，SWFObject（1.5 版本）只替换表情符号容器中的内容。如果没有定义一个 id 属性，则 object 元素将自动继承包含元素的 id。

- 只要 JavaScript 可用，SWFObject（2.0 版本）将整个要替换的内容（包括所引用的 HTML 容器元素）替换为 Flash 版本支持的 Flash 内容。但是，UFO 只替换表情符号容器中的内容。如果没有定义一个 id 属性，则 object 元素将自动继承包含元素的 id。
- SWFObject（2.0 版本）中不包含 UFO 的 setcontainercss 功能。
- 判断 SWFObject（2.0 版本）是否支持 MIME 类型 application / xhtml + xml。
- SWFObject（2.0 版本）不支持 XML MIME 类型，这是由设计决定的，主要因为只有少数 Web 开发人员使用，而且其他主要的浏览器厂商正在寻求一种新的 HTML 解析标准（HTML 5），它与当前的 W3C 解析方法不同。

7.4　Flex 与 JavaScript 的通信

现在我们不仅可以用 ActionScript 编写一个 Flex 程序，还可以通过 swfobject.js 将用 Flex 制作的程序嵌入网页中。但是这样会面临一个问题：程序虽然被嵌入网页中，但是要实现各种交互则需要这两部分相互协调才能完成。比如，要给 Flash Player 发送一条视频流的 URL，怎么做？这就需要用 JavaScript 控制 Flex 程序，或者用 Flex 程序控制 JavaScript。在 Flex 中已经内置了相关方法，可以用来向 JavaScript 发送消息。

7.4.1　使用 Flex 调用 JavaScript 函数

Flex 通过使用原型的 ExternalInterface.call() 函数在 JavaScript 中调用方法

ExternalInterface.call(function_name:string,参数为 string),参数 function_name 为调用 JavaScript 函数的名称，是 JavaScript 函数的必需参数。这个函数有返回值，这意味着 JavaScript 函数可以将结果返回给 ExternalInterface.call()函数调用。先举一个简单的例子——JavaScript 函数：

```
function sayHelloFromJs(message)
{
    alert(message);  //message 是由 Flex 端传过来的
    return "echo from javascript:" + message; //返回给 Flex 端的消息
 }
```

Flex 的调用：

```
var str:String = ExternalInterface.call("sayHelloFromJs","Hello,Javascript.");
Alert.show(str); //显示 JavaScript 端返回的消息
```

这样，一个简单的通信连接就完成了。

7.4.2 使用 JavaScript 调用 Flex 函数

在 Flex 端需要声明和页面交互的 JavaScript 方法。通过 ExternalInterface 类的 addCallback()函数即可实现。

addCallback()函数的原型为 addCallback(js_function:String, flex_function:Function),第一个参数 js_function 是 JavaScript 可以调用的方法名称，第二个参数 flex_function 是 JavaScript 回调的 Flex 方法。下面举一个简单的例子。

在Flex 端：

```
public function sayHelloFromFlex(message:String):String
{
     Alert.show(message); //JavaScript 端传过来的消息
     var str:String = "echo from flex:" + message;
     return str; //返回给 Flex 端的消息
}
public function initApp()
{
      ExternalInterface.addCallback("sayHelloFromFlex",sayHelloFromFlex);//注册与页面交互的方法
}
```

在 JavaScript 端，首先要引用并获取 SWF 对象，如果我们已经拿到了该对象，就可以调取对象中的方法：

```
<script language = 'JavaScript' charset = 'uft-8'>
function testFlexFunc()
{
     var strMessage = MyFlexApp.sayHelloFromFlex("Hello, Flex");
```

```
    alert(strMessage);
}
</script>
<button onclick="testFlexFunc()"> TestFlexFunc</button>
```

这样就实现了 JavaScript 端主动向 Flex 端发送消息的功能了。

7.4.3　使用 JavaScript 获取 SWF 对象的引用

在 JavaScript 代码中，首先要定义一个 object 标签对象，object 标签对象的大概格式如下：

```
<object type='application/x-shockwave-flash'
 data='FlexAndJs.swf' width='600' height='480'
 name='test' id='MyFlexApps'>
<param name='allowScriptAccess' value='always' />
<param name='movie' value='FlexAndJs.swf' />
<param name='quality' value='high' />
<param name='scale' value='noScale' />
<param name='wmode' value='transparent' />
<embed src="FlexObject.swf" width="640" height="378"
   name ="FlexObject"
   play="true"
   loop="false"
   allowScriptAccess="sameDomain"
   type="application/x-shockwave-flash"
   pluginspage="http://www.adobe.com/go/getflashplayer">
</embed>
</object>
```

定义好这个节点标签后，还需要再写一个 JavaScript 函数来获取该 SWF 对象的引用。

```
function getSWFObject(movieName)
{
   if(document[movieName])
   {
        return document[movieName];
   }else if(window[movieName]){
        return window[movieName];
   }else if(document.embeds && document.embeds[movieName]){
        return document.embeds[movieName];
   }else{
        return document.getElementById(movieName);
   }
}
```

在上面代码中，testFlexFunc()函数调用 Flex 函数的语句可以改写为：

```
var strMessage = getSWFObject["MyFlexApps"].sayHelloFromFlex("Hello, Flex");
```

7.4.4 实例：使用 SWFObject 将 Flash 播放器嵌入网页中

1. Flex 代码

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()"
layout="absolute" width="200" height="160">
<mx:Script>
 <![CDATA[
  import mx.controls.Alert;
  public function sayHelloFromFlex(message:String):String{
   Alert.show(message);
   var str:String = "Echo from Flex: "+ message;
   return str;
  }
  public function initApp():void {
   Security.allowDomain("*");
   Security.allowInsecureDomain("*");
   ExternalInterface.addCallback("sayHelloFromFlex",sayHelloFromFlex);
  }
  public function invokeJsFunc():void {
   var str:String = ExternalInterface.call("sayHelloFromJs","Hello,Javascript.");
   Alert.show(str);
  }
 ]]>
</mx:Script>
<mx:Button x="39" y="68" label="Invoke JS Function" click="invokeJsFunc()"/>
</mx:Application>
```

2. HTML 代码

```
<html>
<head></head>
<body scroll="no">
  <script>
  function sayHelloFromJs(value){
        alert(value);
        return "Echo from Js: " + value;
    }

  function invokeFlexFunc(){
        var message = "Hello,Flex.";
        var str = getSWFObject("MyFlexApps").sayHelloFromFlex(message);
        alert(str);
    }
  function getSWFObject(movieName)
```



```
    {
     if(document[movieName])
     {
      return document[movieName];
     }else if(window[movieName]){
      return window[movieName];
     }else if(document.embeds && document.embeds[movieName]){
      return document.embeds[movieName];
     }else{
      return document.getElementById(movieName);
     }
    }
</script>
  <object type='application/x-shockwave-flash'
 data='FlexAndJs.swf' width='200' height='160'
 name='test' id='MyFlexApps'>
<param name='allowScriptAccess' value='always' />
<param name='movie' value='FlexAndJs.swf' />
<param name='quality' value='high' />
<param name='scale' value='noScale' />
<param name='wmode' value='transparent' />
</object>
<input type="button" value="InvokeFlexFunction" onclick="invokeFlexFunc()"/>
</body>
</html>
```

7.5　播放器的制作

学习了上面的技术，就可以在 Web 端操作 Flex 进行推/拉流了。无论是推流端，还是拉流端，都需要有一套完整的播放器才可以预览或者观看视频。本节介绍如何制作一个功能比较完善的 Flash 播放器。这一部分知识点繁多，涉及各种功能，下面将知识点进行细化，以方便读者阅读。

7.5.1　主要功能

在制作播放器时，往往要根据需求编写各种各样的代码，但基础的功能不变，具体包括以下几项。

- 视频的暂停/播放。
- 视频的拖放播放和定点播放。
- 音量被禁用/打开。
- 拖动滑块来控制音量。

- 视频缓冲区进度突出显示。
- 视频全屏处理：单击按钮或单击视频屏幕实现全屏显示。

7.5.2 相关变量

下面是一些关键变量：

```
private var isPause:Boolean = false;          //暂停状态
private var isSound:Boolean = true;           //声音状态（是否禁音）
private var _volume :Number = 0.6;            //默认音量大小(最大值为1)
private var isFullScreen:Boolean = false;     //是否是全屏
private var totalTime:Number;                 //播放总时间
private var playPosition:Number;              //剪辑位置
private var videoUrl:String;                  //视频文件地址
private var videoWidth:Number;                //视频宽度
private var videoHeight:Number;               //视频高度
private var isAutoPlay:Boolean = true;        //是否自动播放
```

7.5.3 初始化视频画布

打开视频播放页面后，首先初始化视频播放画面，根据接收的用户参数，初始化视频画面的大小。对象定义如下：

```
import mx.events.SliderEvent; //滑块事件命名空间引用

private var nc:NetConnection; //媒体连接对象
private var ns:NetStream;     //网络流对象
private var metaDataObj:Object = {}; //媒体的元数据信息
private var video:Video ;          //视频对象
```

初始化方法如下：

```
private function init():void
{
    videoUrl = parameters.videoUrl;
    videoWidth = parameters.videoWidth;
    videoHeight = parameters.videoHeight
    video = new Video(videoWidth,videoHeight);
    video.smoothing = true;//画面平滑处理，去掉全屏后的水纹，以提高画面清晰度
    uic.addChild(video); //将视频对象添加到页面
}
```

7.5.4 加载视频流并播放

当视频初始化时，视频播放方法被调用，并且该方法被放置在应用程序事件的标题中。

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
```


```
                xmlns:s="library://ns.adobe.com/flex/spark"
                xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955"
minHeight="600" initialize="init()" creationComplete="startVideo()">

    private function startVideo():void
    {

        nc = new NetConnection();
        nc.addEventListener(NetStatusEvent.NET_STATUS,netStatusHandler);  //添加播放连接
监听事件
        nc.connect(null);

    }
```

当连接对象连接成功时播放视频。上面代码中的 nc.connect(null);表示如果不使用 Flash Media Server，则可以使用 null 作为参数，通过本地文件系统或 Web 服务器播放视频和 MP3 文件。

```
    private function netStatusHandler(e:NetStatusEvent):void
    {
        ns = new NetStream(nc);
        metaDataObj.onMetaData = this.onMetaData;
        ns.client = metaDataObj;
        video.attachNetStream(ns);
        //ns.bufferTime = 5;
        ns.play(videoUrl);
        soundProcess.value = _volume;
        soundTrans.volume = _volume
        ns.soundTransform = soundTrans;
        this.addEventListener(Event.ENTER_FRAME,EnterFrameHandler); //添加播放过程中的监
听事件
        ns.addEventListener(NetStatusEvent.NET_STATUS,NetStreamStatusHandler);//添加播
放完毕(或其了状态)后的监听事件

        if(!isAutoPlay) //客户端没有设置为自动播放时的处理
        {
            ns.pause();
            btnPlay.source = playClass;
            isPause = true;
        }
        else
        {
            btnPlay.source = pauseClass;
            isPause = false;
        }
    }
    //获取视频的元数据信息，这里的元数据信息包括视频编码、视频码率、音频编码、音频码率、音/视频文件大
```


```
小、流文件总大小、播放总时间等
    private function onMetaData(obj:Object):void
    {
        totalTime = obj.duration;
        fileSize = (obj.filesize/(1024*1024)).toFixed(2).toString()+"MB";//将单位换算成
MB 并保留两位小数
    }
```

说明：

- this.addEventListener（Event.ENTER_FRAME，EnterFrameHandler）：用于监视播放过程中的事件处理。由于播放进度和缓存进度是实时显示的，以及在某个时间点还需要动态实时渲染给用户，所以当视频进入加载画面时，需要实时监控。
- ns.addEventListener（NetStatusEvent.NET_STATUS，NetStreamStatusHandler）：用于在视频流播放后监听事件处理。
- onMetaData：是一种回调方法，可以在客户端加载到视频流中时，异步检索有关媒体的元数据，如总媒体大小、总播放时间、采样率等。

7.5.5 高亮显示播放进度及缓冲进度

```
    //播放进度和缓冲进度处理
    private function EnterFrameHandler(e:Event):void
    {

        if (totalTime>0)
        {
            playTime = ns.time;// ns.time 为流媒体实时播放的时间
        }

        if (ns.bytesLoaded>0)
        {
            bufferRect.width = ns.bytesLoaded / ns.bytesTotal*(playProcess.width-10);//
计算缓冲方框的宽度(滑块本身也有一定的宽度，减去约 10px 宽度)
        }

    }
```

说明：

- 这里将 playTime 作为播放进度条中当前实时播放的时间点，视频的总时间作为播放进度条显示的最大值。

```
    <mx:HSlider id="playProcess"minimum="0" value="{playTime}" maximum="{totalTime}"/>
```

- ns.bytesLoaded 是缓存的流媒体字节的大小（以字节为单位），ns.bytesTotal 是流媒体的

总大小，缓存大小的比例可以在播放进度条的相应位置上进行绘制。缓存进度 Strip 实际上是一个矩形框，可以放在播放进度条的下面，初始宽度为 0，当缓冲区达到 100% 时，即缓冲完成，缓冲区长度和播放进度条长度相等。缓冲方框可以用 BorderContainer 来制作，具体代码如下：

```
    <s:BorderContainer x="14" y="411" width="0" height="4" id="bufferRect"
buttonMode="true" borderColor="#70b2ee" backgroundColor="#70b2ee">
        </s:BorderContainer>
```

页面所有的控件和标签如下：

```
    <mx:Image source="{playClass}" click="play();" id="btnPlay" buttonMode="true" x="16"
y="423"/>
        <mx:UIComponent id="uic" height="400" width="640" click="play()"
doubleClickEnabled="true" doubleClick="display()" buttonMode="true" y="0" x="0"/>
        <s:BorderContainer x="14" y="411" width="0" height="4" id="bufferRect"
buttonMode="true" borderColor="#70b2ee" backgroundColor="#70b2ee">
        </s:BorderContainer>
        <mx:Label text="{formatTime(playTime)}/{formatTime(totalTime)}{fileSize}"
width="150" x="60" y="427"/>
        <mx:HSlider id="playProcess"  minimum="0" value="{playTime}"
maximum="{totalTime}" change="play_onchange(event)" thumbPress="thumbPress();"
thumbRelease="thumbRelease();"
                    alpha="0.5"    dataTipFormatFunction="dataTipFormat"
buttonMode="true" showTrackHighlight="true" x="9" y="398" width="630"/>

        <mx:Image source="{sound1}" click="closeSound();" id="soundImg"
buttonMode="true" x="364" y="421"/>
        <mx:HSlider id="soundProcess" x="406" y="420"  minimum="0" maximum="1"
change="sound_thumbChanges(event)"
                    showTrackHighlight="true"
dataTipFormatFunction="SoundTipFormat"   buttonMode="true"  width="135"/>
        <mx:Button label="全屏" click="display();" buttonMode="true" cornerRadius="20"
labelPlacement="right" paddingLeft="6" x="561" y="423"/>
        <mx:SWFLoader id="load" x="16" y="486"/>
```

7.5.6　视频的播放与暂停

视频的暂停与播放会调用视频流的 pause()方法和 resume()方法，通过是否暂停的状态变量进行控制，部分代码如下：

```
    //播放、暂停设置
    private function play():void
    {
        if(isPause)
        {
```


```
            ns.resume();
        btnPlay.source = pauseClass; //设置按钮图标为点击暂停图标
            isPause = false;

        }
        else
        {
            ns.pause();
            btnPlay.source = playClass; //设置按钮图标为点击播放图标
            isPause = true;
        }
    }
```

7.5.7 拖曳滑块播放视频

拖曳滑块播放视频，主要用于确定并记录流的剪辑位置。要找到剪辑的最终位置，可以调用视频流 seek（参数）方法，其中参数为当前剪辑位置。如果不直接单击而是拖曳滑块，那么剪辑的最终位置应该是鼠标左键弹起的位置。此时的单击相当于实际触发滑块的移动事件，滑动快速移动到单击的位置，部分代码如下：

```
    //拖动进度条时改变播放位置
    private function play_onchange(event:SliderEvent):void
    {
        if(ns.time == 0)
        {
            playProcess.value = 0;
            return;
        }
        playPosition = playProcess.value;                    //保证播放进度统一
        ns.seek(playPosition);
    }

    //进度条鼠标按下
    private function thumbPress():void
    {
        ns.pause();
    }
    //进度条鼠标弹起，指拖动滑块时鼠标弹起
    private function thumbRelease():void
    {
        //ns.seek(playPosition);
         btnPlay.source = pauseClass;
        isPause = false;
        ns.resume();
    }
```

7.5.8　播放结束处理

正常播放完视频后，播放指针归零，即播放进度条上的滑块指向开始的位置，播放按钮处于准备就绪状态，视频流处于暂停状态。可以通过视频流的当前状态信息来判断视频是否播放结束。例如，在下面的代码中，可以通过 e.info.code 状态值获得各种状态，代码片段如下：

```
//播放完毕处理
private function NetStreamStatusHandler(e:NetStatusEvent):void
{
    if(e.info.code == "NetStream.Play.Stop")
    {
        ns.seek(0);
        btnPlay.source = playClass;
        isPause = true;
        ns.pause();
    }
}
```

7.5.9　音量大小控制

视频声音控制通过 SoundTransform 类实现，该类包含音量和平移的属性。如果静音后运行调节滑块，则需要再定义一个临时变量 tmpSound，以便开启声音时为最终设置的音量。

```
//静音、开音控制
private function closeSound():void
{
    if(isSound)
    {
        soundImg.source = sound;
        tmpSound= ns.soundTransform;
        soundTrans.volume = 0;
        ns.soundTransform = soundTrans; // 这里禁音直接使用 ns.soundTransform.volume =
0 是不行的，需要用对象赋值
        isSound = false;
    }else
    {
        soundImg.source = sound1;
        ns.soundTransform = tmpSound;
        isSound = true;
    }
}

//通过滑块调整声音
private function sound_thumbChanges(event:SliderEvent):void
{
    tmpSound.volume = soundProcess.value;
```

```
        ns.soundTransform = tmpSound;
    }
```

7.5.10 全屏显示控制

可以使用 FlashCanvas 对象的 stage 属性值来设置视频全屏显示，但是考虑到正常的屏幕和宽屏处理，常见的显示分辨率可以分为 4：3（1024×768），5：4（1280×1024）），16：9 和 16：10（这里为宽屏测试，需要后续处理）。单击全屏按钮或单击屏幕全屏按钮，可以调用 display() 方法，代码片段如下：

```
     //切换全屏显示
    private function display():void
    {
        if(!isFullScreen)
        {
            stage.fullScreenSourceRect = new Rectangle(video.x,
video.y,video.width,video.height);
            stage.displayState =StageDisplayState.FULL_SCREEN;
            isFullScreen = true;
        }else
        {
            stage.displayState = StageDisplayState.NORMAL;
            isFullScreen = false;
        }

    }
```

7.5.11 流数据字符格式化

视频处于播放状态时，当前时间和总时间是以秒（s）为单位的，比如 180s 的文件，当播放到一半时显示 90s，这时需要按时间格式来显示才显得友好。另外，还有音量的值是介于 0~1 的某个值，也需要按百分比来显示才显得更友好。代码片段如下：

```
     //格式化时间
    private function formatTime(time:Number):String
    {
        var min:Number = Math.floor(time/60);
        var sec:Number = Math.floor(time%60);
        var timeResult:String = (min < 10 ? "0"+min.toString() : min.toString()) + ":"
+ (sec == 10 ? "0"+sec.toString() : sec.toString());
        return timeResult;
    }
    //声音 slider 格式化
    private function SoundTipFormat(data:Number):String
    {
```

```
        return  (data*100).toFixed(0).toString()+"%"; //拖动声音进度条时显示百分比音量
    }
```

7.5.12　视频画面的平滑优化处理

一般视频在全屏显示后，会让文字或图像产生失真的感觉，产生水纹。对于这个问题的处理，Flex 封装了一个简单、有效的方法，只需要设置一个属性即可，即在 video 对象中设置一个属性：

```
video.smoothing = true;
```

把该属性值设置为 true，表示启用画面优化处理，并且这个设置能大大提高画面的质量。

7.5.13　播放接口的调用

代码采用 Flex 开放的播放器编译后，最终生成的是一个.swf 文件，其需要通过页面加载来调用，可以是静态的 HTML 页面，也可以是动态的 ASPX 页面，在调用过程中会引用 swfobject.js 文件：

```
<script type="text/javascript" src="swfobject.js"></script>
<script type="text/javascript">
    var swfVersionStr = "10.0.0";
    var xiSwfUrlStr = "playerProductInstall.swf";
    var flashvars = {};
    flashvars.videoUrl = "xxx.flv";
    flashvars.videoWidth = "640";
    flashvars.videoHeight = "400";
    flashvars.videoVolumn = "0.6";
    flashvars.isAutoPlay = "true";
    var params = {};
    params.quality = "high";
    params.bgcolor = "#ffffff";
    params.allowscriptaccess = "sameDomain";
    params.allowfullscreen = "true";
    var attributes = {};
    attributes.id = "player";
    attributes.name = "player";
    attributes.align = "middle";
    swfobject.embedSWF(
        "player.swf", "flashContent",
        "650", "450",
        swfVersionStr, xiSwfUrlStr,
        flashvars, params, attributes);
</script>

<div id="flashContent"></div>
```

其中 flashContent 标签为嵌入的 SWF 播放器的容器。

7.5.14 实例：制作自定义播放器

```
    <?xml version="1.0" encoding="utf-8"?>
    <s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
                   xmlns:s="library://ns.adobe.com/flex/spark"
                   xmlns:mx="library://ns.adobe.com/flex/mx" fontSize="12"
initialize="init();" backgroundAlpha="0.6" backgroundColor="#FFFFFF">

    <fx:Declarations>
    <!-- 将非可视元素（例如服务、值对象）放在此处 -->
    </fx:Declarations>

    <fx:Script>
    <![CDATA[
        import mx.controls.Alert;
        import mx.events.MetadataEvent;
        import mx.events.SliderEvent;
        import mx.controls.sliderClasses.Slider;
        import mx.events.VideoEvent;
        import mx.utils.ObjectUtil;
        import mx.controls.ProgressBarDirection;

    [Embed(source="images/play.png")]
    [Bindable]
    private var playClass:Class;                    //播放图标样式

    [Embed(source="images/pause.png")]
    [Bindable]
    private var pauseClass:Class;                   //暂停图标样式

    [Embed(source="images/sound1.png")]
    [Bindable]
    private var sound1:Class;                       //声音样式 1

    [Embed(source="images/sound.png")]
    [Bindable]
    private var sound:Class;                        //声音样式 2（静音）

    [Bindable]
    private var videoSource:String;                 //媒体路径

    private var isPause:Boolean = false;            //暂停状态
    private var isSound:Boolean = true;             //声音状态
    private var isFullScreen:Boolean = false;       //是否是全屏
```

```
private var tmpSound:Number = 0;                    //临时声音大小

[Bindable]
private var fileSize:String;                        //视频文件大小

[Bindable]
private var playPosition:Number;                    //播放进度

private function init():void{
    videoSource = parameters.videosource;
}

private function playingMove(event:VideoEvent):void{
    my_hs.value = myVid.playheadTime;               //视频播放时同步进度条状态值

}

//拖动进度条时改变播放位置
private function hs_onchange(event:SliderEvent):void{
    if(myVid.playheadTime == -1){
        my_hs.value = 0;
        return;
    }
    playPosition = my_hs.value;                     //播放进度统一
    myVid.playheadTime = playPosition;
}

//进度条按下
private function thumbPress():void{
    myVid.pause();
}
//进度条弹起
private function thumbRelease():void{
    myVid.playheadTime = playPosition;
    myVid.play();
}


//播放，暂停
private function playButton():void{
    if(!isPause){
        myVid.play();
        playBtn.source = pauseClass;
        isPause = true;
    }else{
        myVid.pause();
        playBtn.source = playClass;
```

```
            isPause = false;
        }
    }

    //播放完成
    private function vidComplete():void{
        playBtn.source = playClass;
        isPause = false;
    }

    //停止播放
    private function stopButton():void{
        myVid.stop();
        playBtn.source = playClass;
        isPause = false;
    }

    //切换全屏显示
    private function display():void{
        if(!isFullScreen){
            stage.fullScreenSourceRect = new
Rectangle(myVid.x,myVid.y,myVid.width,myVid.height);
            stage.displayState =StageDisplayState.FULL_SCREEN;
            isFullScreen = true;
        }else{
            stage.displayState = StageDisplayState.NORMAL;
            isFullScreen = false;
        }
    }

    //调整声音
    private function sound_thumbChanges(event:SliderEvent):void{
        myVid.volume = hs_sound.value;
    }

    //静音
    private function closeSound():void{
        if(isSound){
            closeImg.source = sound;
            tmpSound = myVid.volume;
            myVid.volume = 0;
            isSound = false;
        }else{
            closeImg.source = sound1;
            myVid.volume = tmpSound;
            isSound = true;
        }
```

```
    }


    //格式化时间
    private function formatTime(time:Number):String{
        var min:Number = Math.floor(time/60);
        var sec:Number = Math.floor(time%60);
    var timeResult:String = (min < 10 ? "0"+min.toString() : min.toString()) + ":" + (sec
== 10 ? "0"+sec.toString() : sec.toString());
    return timeResult;
    }
    //slider 格式化
    private function dataTipFormat(data:Number):String{
        return formatTime(data);
    }


    private function myVid_metadataReceived(evt:MetadataEvent):void {

        var meta:Object = evt.info; // 视频的元数据信息
        fileSize = (meta.filesize/(1024*1024)).toFixed(2).toString()+"MB";//单位换算成 MB
并保留两位小数

        /* 读取所有元数据属性和值
        var i:int = 0;
        var arr:Array = [];
        var item:String;
         var value:*;
        for (item in meta) {
            if (ObjectUtil.isSimple(meta[item])) {
                if (meta[item] is Array) {
                    value = "[Array]";
                } else {
                    value = meta[item]
                }

                Alert.show(item+":"+value);
            }

        }*/


    }
```

```
    ]]>



    </fx:Script>

    <s:BorderContainer borderColor="#66ccff" y="0" cornerRadius="0" borderWeight="0"
borderVisible="true" dropShadowVisible="false" x="0" width="660">

    <mx:VideoDisplay id="myVid" y="10" height="400" width="640"  source="{videoSource}"
autoPlay="false"  buttonMode="true"  click="playButton();"  ready="myVid.visible =
true;" metadataReceived="myVid_metadataReceived(event);"
      playheadUpdate="playingMove(event)"  complete="vidComplete();"
doubleClickEnabled="true"  doubleClick="display();" contentBackgroundAlpha="1.0"
contentBackgroundColor="#000000" x="10"

    />

    <mx:HBox width="640" verticalAlign="middle" x="10" y="415" height="90">
       <mx:Image source="{playClass}" click="playButton();" id="playBtn"
buttonMode="true"/>
       <mx:Label
text="{formatTime(myVid.playheadTime)}/{formatTime(myVid.totalTime)}
 {fileSize}" width="150"/>
       <mx:HRule height="0" width="200" buttonMode="true"/>
       <mx:Image source="{sound1}" click="closeSound();" id="closeImg"
buttonMode="true"/>
       <mx:HSlider width="100" id="hs_sound" minimum="0" maximum="1"
                  change="sound_thumbChanges(event)"
                  value="{myVid.volume}" buttonMode="true" />
       <mx:Button label="全屏" click="display();" buttonMode="true" cornerRadius="20"
labelPlacement="right" paddingLeft="6"/>
    </mx:HBox>
    <mx:HSlider width="640" id="my_hs" minimum="0" maximum="{myVid.totalTime}"
height="10" showTrackHighlight="true" buttonMode="true" liveDragging="true"
              change="hs_onchange(event)" thumbPress="thumbPress();"
thumbRelease="thumbRelease();" x="10" y="422"  dataTipFormatFunction="dataTipFormat" />



    </s:BorderContainer>
    </s:Application>
```

7.6　Web 端开发实战

掌握了前面的技术知识，下面就可以编写应用实例了。下面基于 swfobject.js，编写并封装一套 SDK，方便我们在 HTML 中进行调用与配置。另外，对于 Flex 播放器，也可根据实际功能的需要进行自定义编写。

7.6.1　推流

1. HTML

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh_cn" lang="zh_cn">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>simplest_as3_RTMP_streamer</title>
<script language="JavaScript" type="text/javascript">
<!--
//v1.7
// Flash Player Version Detection
// Detect Client Browser type
// Copyright 2005-2008 Adobe Systems Incorporated.  All rights reserved.
var isIE  = (navigator.appVersion.indexOf("MSIE") != -1) ? true : false;
var isWin = (navigator.appVersion.toLowerCase().indexOf("win") != -1) ? true : false;
var isOpera = (navigator.userAgent.indexOf("Opera") != -1) ? true : false;
function ControlVersion()
{
      var version;
      var axo;
      var e;
      // NOTE : new ActiveXObject(strFoo)如果 strFoo 没有注册就抛出异常
      try {
           // 版本设置为 7.X 或更高版本
           axo = new ActiveXObject("ShockwaveFlash.ShockwaveFlash.7");
           version = axo.GetVariable("$version");
      } catch (e) {
      }
      if (!version)
      {
           try {
                // 版本只能设置为 6.X
                axo = new ActiveXObject("ShockwaveFlash.ShockwaveFlash.6");

                // 已安装的播放器是 6.0 的一些修订版
                // GetVariable("$version") 在 6.0.22 ~ 6.0.29 版本中被禁用
                // 所以此处必须小心
```

```
                // 默认为第一个公开版本
                version = "WIN 6,0,21,0";
            // 如果 AllowScripAccess 不存在就抛出 (6.0r47 推出)
            axo.AllowScriptAccess = "always";
                // 在 6.0r47 以上版本使用更安全
                version = axo.GetVariable("$version");
            } catch (e) {
            }
        }
        if (!version)
        {
            try {
                // 播放器版本将设置为 4.X 或 5.X
                axo = new ActiveXObject("ShockwaveFlash.ShockwaveFlash.3");
                version = axo.GetVariable("$version");
            } catch (e) {
            }
        }
        if (!version)
        {
            try {
                // 播放器版本将设置为 3.X
                axo = new ActiveXObject("ShockwaveFlash.ShockwaveFlash.3");
                version = "WIN 3,0,18,0";
            } catch (e) {
            }
        }
        if (!version)
        {
            try {
                // 播放器版本将设置为 2.X
                axo = new ActiveXObject("ShockwaveFlash.ShockwaveFlash");
                version = "WIN 2,0,0,11";
            } catch (e) {
                version = -1;
            }
        }

        return version;
}
// 需要检测 Flash 播放器插件版本信息
function GetSwfVer(){
        // NS/Opera version >= 3 check for Flash plugin in plugin array
        var flashVer = -1;

        if (navigator.plugins != null && navigator.plugins.length > 0) {
            if (navigator.plugins["Shockwave Flash 2.0"] ||
```


```
navigator.plugins["Shockwave Flash"]) {
                    var swVer2 = navigator.plugins["Shockwave Flash 2.0"] ? " 2.0" : "";
                    var flashDescription = navigator.plugins["Shockwave Flash" +
swVer2].description;
                    var descArray = flashDescription.split("");
               var tempArrayMajor = descArray[2].split(".");
               var versionMajor = tempArrayMajor[0];
               var versionMinor = tempArrayMajor[1];
               var versionRevision = descArray[3];
               if (versionRevision == "") {
                         versionRevision = descArray[4];
                    }
                    if (versionRevision[0] == "d") {
                         versionRevision = versionRevision.substring(1);
                    } else if (versionRevision[0] == "r") {
                         versionRevision = versionRevision.substring(1);
                         if (versionRevision.indexOf("d") > 0) {
                              versionRevision = versionRevision.substring(0,
versionRevision.indexOf("d"));
                         }
                    }
                    var flashVer = versionMajor + "." + versionMinor + "." +
versionRevision;
                }
          }
          // MSN/WebTV 2.6 支持 Flash 4
          else if (navigator.userAgent.toLowerCase().indexOf("webtv/2.6") != -1)
flashVer = 4;
          // WebTV 2.5 supports Flash 3
          else if (navigator.userAgent.toLowerCase().indexOf("webtv/2.5") != -1)
flashVer = 3;
          // older WebTV supports Flash 2
          else if (navigator.userAgent.toLowerCase().indexOf("webtv") != -1) flashVer =
2;
          else if ( isIE && isWin && !isOpera ) {
           flashVer = ControlVersion();
       }
          return flashVer;
    }
    // 当使用 reqmajorver, reqminorver, reqrevision 返回 true（如果版本或更高版本可用）
    function DetectFlashVer(reqMajorVer, reqMinorVer, reqRevision)
    {
          versionStr = GetSwfVer();
          if (versionStr == -1 ) {
               return false;
          } else if (versionStr != 0) {
               if(isIE && isWin && !isOpera) {
```

```
                // 指定 "WIN 2,0,0,11"
                tempArray         = versionStr.split("");// ["WIN", "2,0,0,11"]
                tempString        = tempArray[1];// "2,0,0,11"
                versionArray      = tempString.split(",");// ['2', '0', '0', '11']
            } else {
                versionArray      = versionStr.split(".");
            }
            var versionMajor      = versionArray[0];
            var versionMinor      = versionArray[1];
            var versionRevision   = versionArray[2];

            if (versionMajor > parseFloat(reqMajorVer)) {
                return true;
            } else if (versionMajor == parseFloat(reqMajorVer)) {
                if (versionMinor > parseFloat(reqMinorVer))
                    return true;
                else if (versionMinor == parseFloat(reqMinorVer)) {
                    if (versionRevision >= parseFloat(reqRevision))
                        return true;
                }
            }
            return false;
        }
}
function AC_AddExtension(src, ext)
{
  if (src.indexOf('?') != -1)
    return src.replace(/\?/, ext+'?');
  else
    return src + ext;
}
function AC_Generateobj(objAttrs, params, embedAttrs)
{
  var str = '';
  if (isIE && isWin && !isOpera)
  {
    str += '<object ';
    for (var i in objAttrs)
    {
      str += i + '="' + objAttrs[i] + '" ';
    }
    str += '>';
    for (var i in params)
    {
      str += '<param name="' + i + '" value="' + params[i] + '" /> ';
    }
    str += '</object>';
```

```
    }
    else
    {
      str += '<embed ';
      for (var i in embedAttrs)
      {
        str += i + '="' + embedAttrs[i] + '" ';
      }
      str += '></embed>';
    }
    document.write(str);
}
function AC_FL_RunContent(){
    var ret =
      AC_GetArgs
      ( arguments, ".swf", "movie", "clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
       , "application/x-shockwave-flash"
      );
    AC_Generateobj(ret.objAttrs, ret.params, ret.embedAttrs);
}
function AC_SW_RunContent(){
    var ret =
      AC_GetArgs
      ( arguments, ".dcr", "src", "clsid:166B1BCA-3F9C-11CF-8075-444553540000"
       , null
      );
    AC_Generateobj(ret.objAttrs, ret.params, ret.embedAttrs);
}
function AC_GetArgs(args, ext, srcParamName, classid, mimeType){
    var ret = new Object();
    ret.embedAttrs = new Object();
    ret.params = new Object();
    ret.objAttrs = new Object();
    for (var i=0; i < args.length; i=i+2){
      var currArg = args[i].toLowerCase();
      switch (currArg){
        case "classid":
          break;
        case "pluginspage":
          ret.embedAttrs[args[i]] = args[i+1];
          break;
        case "src":
        case "movie":
          args[i+1] = AC_AddExtension(args[i+1], ext);
          ret.embedAttrs["src"] = args[i+1];
          ret.params[srcParamName] = args[i+1];
          break;
```


```
        case "onafterupdate":
        case "onbeforeupdate":
        case "onblur":
        case "oncellchange":
        case "onclick":
        case "ondblclick":
        case "ondrag":
        case "ondragend":
        case "ondragenter":
        case "ondragleave":
        case "ondragover":
        case "ondrop":
        case "onfinish":
        case "onfocus":
        case "onhelp":
        case "onmousedown":
        case "onmouseup":
        case "onmouseover":
        case "onmousemove":
        case "onmouseout":
        case "onkeypress":
        case "onkeydown":
        case "onkeyup":
        case "onload":
        case "onlosecapture":
        case "onpropertychange":
        case "onreadystatechange":
        case "onrowsdelete":
        case "onrowenter":
        case "onrowexit":
        case "onrowsinserted":
        case "onstart":
        case "onscroll":
        case "onbeforeeditfocus":
        case "onactivate":
        case "onbeforedeactivate":
        case "ondeactivate":
        case "type":
        case "codebase":
        case "id":
          ret.objAttrs[args[i]] = args[i+1];
          break;
        case "width":
        case "height":
        case "align":
        case "vspace":
        case "hspace":
```



```
        case "class":
        case "title":
        case "accesskey":
        case "name":
        case "tabindex":
          ret.embedAttrs[args[i]] = ret.objAttrs[args[i]] = args[i+1];
          break;
        default:
          ret.embedAttrs[args[i]] = ret.params[args[i]] = args[i+1];
      }
    }
    ret.objAttrs["classid"] = classid;
    if (mimeType) ret.embedAttrs["type"] = mimeType;
    return ret;
  }
  // -->
  </script>
  </head>
  <body bgcolor="#999999">
  <!--影片中使用的 URL-->
  <!--影片中使用的文本-->
  <!-- saved from url=(0013)about:internet -->
  <script language="JavaScript" type="text/javascript">
        AC_FL_RunContent(
            'codebase',
'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=10,0,0,0',
            'width', '670',
            'height', '260',
            'src', 'simplest_as3_RTMP_streamer',
            'quality', 'high',
            'pluginspage', 'http://www.adobe.com/go/getflashplayer_cn',
            'align', 'middle',
            'play', 'true',
            'loop', 'true',
            'scale', 'showall',
            'wmode', 'window',
            'devicefont', 'false',
            'id', 'simplest_as3_RTMP_streamer',
            'bgcolor', '#999999',
            'name', 'simplest_as3_RTMP_streamer',
            'menu', 'true',
            'allowFullScreen', 'false',
            'allowScriptAccess','sameDomain',
            'movie', 'simplest_as3_RTMP_streamer',
            'salign', ''
            ); //end AC code
  </script>
```


```
    <noscript>
          <object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version
=10,0,0,0" width="670" height="260" id="simplest_as3_RTMP_streamer" align="middle">
          <param name="allowScriptAccess" value="sameDomain" />
          <param name="allowFullScreen" value="false" />
          <param name="movie" value="simplest_as3_RTMP_streamer.swf" /><param
name="quality" value="high" /><param name="bgcolor" value="#999999" /><embed
src="simplest_as3_RTMP_streamer.swf" quality="high" bgcolor="#999999" width="670"
height="260" name="simplest_as3_RTMP_streamer" align="middle"
allowScriptAccess="sameDomain" allowFullScreen="false"
type="application/x-shockwave-flash"
pluginspage="http://www.adobe.com/go/getflashplayer_cn" />
          </object>
    </noscript>
    </body>
    </html>
```

2. ActionScript

```
    package {
        import flash.display.MovieClip;
        import flash.net.NetConnection;
        import flash.events.NetStatusEvent;
        import flash.net.NetStream;
        import flash.media.Video;
           import flash.media.Camera;
        import flash.media.Microphone;
          //import flash.media.H264Profile;
        //import flash.media.H264VideoStreamSettings;

        public class simplest_as3_RTMP_streamer extends MovieClip
        {
                var nc:NetConnection;
                var ns:NetStream;
                var nsPlayer:NetStream;
                var vid:Video;
                var vidPlayer:Video;
                var cam:Camera;
                var mic:Microphone;

                var screen_w:int=320;
                var screen_h:int=240;

                public function simplest_as3_RTMP_streamer()
```

```
        {
                nc = new NetConnection();
                nc.addEventListener(NetStatusEvent.NET_STATUS, onNetStatus);
                nc.connect("RTMP://localhost/live");
        }

            private function onNetStatus(event:NetStatusEvent):void{
                trace(event.info.code);
                if(event.info.code == "NetConnection.Connect.Success"){
                    publishCamera();
                    displayPublishingVideo();
                    displayPlaybackVideo();
            }
        }

        private function publishCamera() {

            //Cam

            cam = Camera.getCamera();

            /**
              * public function setMode(width:int, height:int, fps:Number,
favorArea:Boolean = true):void
              *  width:int——宽度，以像素为单位，默认值是 160
              *  height:int——高度，以像素为单位，默认值是 120
              *  fps:Number——帧率，每秒帧数，默认值是 15
              */
            cam.setMode(640, 480, 15);

            /**
             * public function setKeyFrameInterval(keyFrameInterval:int):void
             * 传输完整的视频帧数（称为帧）而不是被视频压缩算法的插值
             * 默认值是 15，这意味着每 15 帧是关键帧。值为 1 意味着每 1 帧是关键帧
             * 允许的值为 1~300
             */
             cam.setKeyFrameInterval(25);

            /**
             * public function setQuality(bandwidth:int, quality:int):void
              * bandwidth:int——指定当前传出视频馈送可以使用的最大带宽量，以字节/秒为单位。
              *    为了指定视频可以使用所需的带宽来保持质量的值，带宽通过
              *    默认值是 16384
              * quality:int——一个整数，指定所需的图片质量级别，由压缩量决定
              *     应用于每个视频帧。可接受的值范围从 1（最低质量，最大压缩）到 100，
              *    （最高质量，无压缩）。指定图片质量可以根据需要而变化，以避免超出带宽
              */
```

```
            cam.setQuality(200000, 90);

            /**
             * public function setProfileLevel(profile:String, level:String):void
             * 设置视频编码的配置文件和级别
             * 轮廓可能值是 h264profile.baseline 和 h264profile.main。默认值是
h264profile.baseline
             * 其他值将被忽略并导致错误
             * 支持的级别是 1、1b、1.1、1.2、1.3、2、2.1、2.2、3、3.1、3.2、4、4.1、4.2、、5
和 5.1
             * 如果分辨率和帧速率有要求，则水平可能会增加
             */
            //var h264setting:H264VideoStreamSettings = new
H264VideoStreamSettings();
             // h264setting.setProfileLevel(H264Profile.MAIN, 4);


            //Mic

            mic = Microphone.getMicrophone();

            /*
             * 编码后的语音质量采用 Speex 的时候，可能的值是 0~10。默认值是 6
             * 更高的数字表示更高的质量，但需要更多的带宽，如下表所示
             * 这是上市的比特率值代表净比特率，不包括打包的开销
              * ----------------------------------------
             * Quality value | Required bit rate (Kbps)
             *----------------------------------------
             *      0        |       3.95
             *      1        |       5.75
             *      2        |       7.75
             *      3        |       9.80
             *      4        |       12.8
             *      5        |       16.8
             *      6        |       20.6
             *      7        |       23.8
             *      8        |       27.8
             *      9        |       34.2
             *      10       |       42.2
             *----------------------------------------
             */
            mic.encodeQuality = 9;

             /* 麦克风捕捉声音的速率，单位为 kHz。可接受的值分别为 5、8、11、22 和 44。默认值是 8kHz
              * 如果声音捕获设备支持此值。否则，默认值是 8kHz 以上的下一个可用捕获电平
              * 声音捕捉设备支持，通常是 11kHz
              *
```

```
            */
            mic.rate = 44;


            ns = new NetStream(nc);
            //H.264 Setting
            //ns.videoStreamSettings = h264setting;
            ns.attachCamera(cam);
            ns.attachAudio(mic);
            ns.publish("myCamera", "live");
        }

        private function displayPublishingVideo():void {
            vid = new Video(screen_w, screen_h);
            vid.x = 10;
            vid.y = 10;
            vid.attachCamera(cam);
            addChild(vid);
        }

        private function displayPlaybackVideo():void{
            nsPlayer = new NetStream(nc);
            nsPlayer.play("myCamera");
            vidPlayer = new Video(screen_w, screen_h);
            vidPlayer.x = screen_w + 20;
            vidPlayer.y = 10;
            vidPlayer.attachNetStream(nsPlayer);
            addChild(vidPlayer);
        }
    }
}
```

7.6.2　拉流

1. HTML

这一部分内容和推流端大同小异，所以就不再举例，重点介绍 ActionScript 部分。

2. ActionScript

```
package {
    import flash.display.Sprite;
    import flash.net.NetConnection;
    import flash.events.NetStatusEvent;
    import flash.events.AsyncErrorEvent;
    import flash.net.NetStream;
```


```
import flash.media.Video;

public class simplest_as3_RTMP_player_multiscreen extends Sprite
{
    var nc0:NetConnection;
    var ns0:NetStream;
        var video0:Video;

        var nc1:NetConnection;
    var ns1:NetStream;
        var video1:Video;

        var nc2:NetConnection;
    var ns2:NetStream;
        var video2:Video;

        var nc3:NetConnection;
    var ns3:NetStream;
        var video3:Video;

        //var border:int=10;
        var app_w:int=640;
        var app_h:int=480;
        var screen_w:int=app_w/2;
        var screen_h:int=app_h/2;

    public function simplest_as3_RTMP_player_multiscreen()
    {
         nc0 = new NetConnection();
         nc0.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler0);
            //HKS | RTMP://live.hkstv.hk.lxdns.com/live/hks
            nc0.connect("RTMP://live.hkstv.hk.lxdns.com/live");

            nc1 = new NetConnection();
        nc1.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler1);
            //Beijing Mobile | RTMP://www.bj-mobiletv.com:8000/live/live1
            nc1.connect("RTMP://www.bj-mobiletv.com:8000/live");

            nc2 = new NetConnection();
        nc2.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler2);
            //Dongguan TV | RTMP://ftv.sun0769.com/dgrtv1/mp4:b1
            nc2.connect("RTMP://ftv.sun0769.com/dgrtv1");

            nc3 = new NetConnection();
        nc3.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler3);
            //CNR Mall | RTMP://wx.cnrmall.com/live/flv
            nc3.connect("RTMP://wx.cnrmall.com/live");
```


```
            }


            private function netStatusHandler0(event:NetStatusEvent):void
            {
                    trace("[video 0] event.info.level: " + event.info.level + "\n",
"event.info.code: " + event.info.code);
                switch (event.info.code)
                {
                    case "NetConnection.Connect.Success":
                          doVideo0(nc0);
                          break;
                   }
            }

                private function netStatusHandler1(event:NetStatusEvent):void
            {
                     trace("[video 1] event.info.level: " + event.info.level + "\n",
"event.info.code: " + event.info.code);
                switch (event.info.code)
                {
                    case "NetConnection.Connect.Success":
                          doVideo1(nc1);
                          break;
                   }
            }

                private function netStatusHandler2(event:NetStatusEvent):void
            {
                     trace("[video 2] event.info.level: " + event.info.level + "\n",
"event.info.code: " + event.info.code);
                switch (event.info.code)
                {
                    case "NetConnection.Connect.Success":
                          doVideo2(nc2);
                          break;
                   }
            }

                private function netStatusHandler3(event:NetStatusEvent):void
            {
                     trace("[video 3] event.info.level: " + event.info.level + "\n",
"event.info.code: " + event.info.code);
                switch (event.info.code)
                {
                    case "NetConnection.Connect.Success":
```

```
                doVideo3(nc3);
                break;
            }
        }


        // play a recorded stream on the server
        private function doVideo0(nc:NetConnection):void {
            ns0 = new NetStream(nc);
            ns0.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler0);

            video0 = new Video(screen_w,screen_h);
                video0.x=0;
                video0.y=0;
            video0.attachNetStream(ns0);

            ns0.play("hks");
            addChild(video0);
        }

        private function doVideo1(nc:NetConnection):void {
            ns1 = new NetStream(nc);
            ns1.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler1);

            video1 = new Video(screen_w,screen_h);
                video1.x=screen_w;
                video1.y=0;
            video1.attachNetStream(ns1);

            ns1.play("live1");
            addChild(video1);
        }

            private function doVideo2(nc:NetConnection):void {
            ns2 = new NetStream(nc);
            ns2.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler2);

            video2 = new Video(screen_w,screen_h);
                video2.x=0;
                video2.y=screen_h;
            video2.attachNetStream(ns2);

            ns2.play("mp4:b1");
            addChild(video2);
        }

            private function doVideo3(nc:NetConnection):void {
```


```
            ns3 = new NetStream(nc);
            ns3.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler0);

            video3 = new Video(screen_w,screen_h);
               video3.x=screen_w;
               video3.y=screen_h;
            video3.attachNetStream(ns3);

            ns3.play("flv");
            addChild(video3);
        }
    }
}
```

7.7　本章小结

经过前面的介绍，相信读者大致了解了 Web 端推流的方法，其大致流程如下：

- 安装 FlashBuilder 并创建一个 Flex 项目。
- 使用 ActionScript 编写一个推流器/拉流器。
- 将 Flex 生成的 SWF 播放器文件通过 swfobject.js 嵌入 HTML 中。
- 利用 JavaScript 与播放器的通信控制推流、拉流与播放等。

在实际中，应该将常用的 JavaScript 功能封装成一套 SDK 以方便使用，下面是笔者封装的一些常用属性以及方法名称：

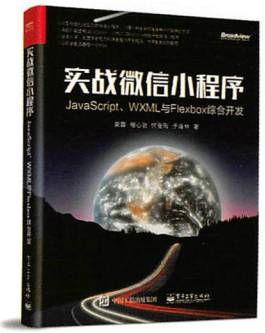
```
initPublish = {
              previewWindowWidth: 862,//视频预览宽度，默认为 862px
              previewWindowHeight: 446,//视频预览高度，默认为 446px
              videoWidth: 640,//视频推流分辨率宽度
              videoHeight: 480,//视频推流分辨率高度
              fps: 15,//帧数
              bitrate: 600,//比特率
              wmode: "transparent",//Flash 显示模式
              quality: "high",//Flash 显示质量
              allowScriptAccess: "always"//允许 Flash 跨域
}

start = function() {...} //开始拉流
stop = function() {...} //停止拉流
startPublish = function() {...} //开始推流
stopPublish = function() {...} //停止推流
getUrl = function() {...} //获取流地址
getCamera = function(...) {...} //获取本地摄像头
```



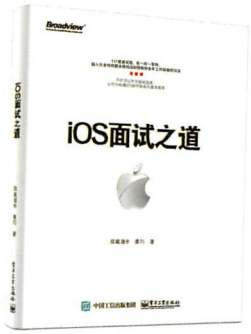
```
getMicro = function() {...} //获取本地麦克风
getFlexCall = function() {...} //接受 Flex 的消息
sentFlex = function() {...} //向 Flex 发送消息
//……
```

实际的需求千变万化，在编写 SDK 时要随机应变，总结出一套最适合项目需求的 SDK。

好书分享



《实战微信小程序：JavaScript、WXML与Flexbox综合开发》
ISBN 978-7-121-31314-1



《iOS面试之道》
ISBN 978-7-121-34262-2

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

第一部分
Nginx介绍

Nginx概述及作用
为什么选择Nginx
安装、配置及使用Nginx

本书内容

第二部分
Nginx-rtmp-module介绍

Nginx-rtmp-module基础
Nginx-rtmp-module进阶
Nginx-rtmp-module应用

第三部分
多终端解决方案

Android端解决方案
iOS端解决方案
Web端解决方案

本书读者

- ☑ 对直播系统开发有兴趣的人员
- ☑ 高级语言开发者
- ☑ 音/视频开发者

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：王　静　欢迎投稿：wangj@phei.com.cn
封面设计：吴海燕

上架建议：程序开发、视频开发

ISBN 978-7-121-35178-5

定价：69.00元